FOR EVERYTHING IN ELECTRONICS

Wireless world

MAY 1983

80p.

Forth microcomputer

UOSAT data and picture decoder

Microprocessor d.v.m.

Versatile accelerometer

Australia	A$ 2.70
Canada	C$ 3.25
Denmark	DKR. 33.25
Germany	DM. 7.00
Greece	DRA. 190.00
Holland	DFL. 8.50
Italy	L 3700
New Zealand	NZ$ 3.50
Norway	NKR. 26.00
Singapore	M$ 5.50
Spain	PTS 250.00
Switzerland	FR. 7.00
U.S.A.	$3.75

Forth computer

With applications ranging from video games to research and process control this microcomputer combines the powers of Forth, a fast, threaded computer language/operating system, with an eight-bit processor having 16 bit internal architecture.

Today a home or personal computer can more than match at lower cost the performance of a typical mid-1970s minicomputer. Then a minicomputer costing tens of thousands of pounds would have a memory of less than 32K words with a cycle time of about a microsecond and a Teletype terminal capable of ten characters per second. Disc-drive memory was rare and expensive and double-precision/floating-point instructions would be executed by software. This microcomputer design costing a few hundred pounds has a 48K read/write memory operating at 666ns, further 8Kbyte rom containing the operating system, a 100-character-per-second terminal and a 200Kbyte disc memory.

	Forth computer	**1970s minicomputer**
Memory size	56K (48K ram, 8K rom)	64K ram
Memory speed	666ns	960ns
C.p.u. 16-bit add time	4.6μs	1.96μs
Output peripherals	composite video RS232 8 ports	64 ports
Input peripherals	parallel keyboard RS232 8 ports	standard peripherals
Disc storage	200Kbyte/drive	5Mbyte/drive
Access time	333ms	35ms
Cost	£100-500	£10,000-100,000

The cost of developing control software and language application packages is the main reason why low-cost microprocessors have not destroyed the minicomputer industry. It will be a long time before any microprocessor has the software support of the PDP11! Further, when designing a home computer from the i.cs upwards one does not have the support of other computers to develop the software on and one cannot afford to develop the software alone. For these reasons the control program was chosen from those already available. This also applied to the choice of language; I was not willing to start from the bottom with machine code, for one sees too little reward for the effort of keying in programs on a hexadecimal keypad, nor was I prepared to design a 'bootstrap' rom that loaded the operating system in from disc, for I felt it an unnecessary commitment while the rest of the system was unproven.

Language/operating-system choice

The most popular operating system and language in the microcomputer field are CP/M and Basic respectively. Although Basic is readily available, in for example the INS8298 rom for the 8080, I was not prepared to use the language for reasons too many to mention but summed up by Dijkstra[1] who said "It is practically impossible to teach good programming to students that have had prior exposure to Basic." He seems equally impressed by most other languages, including Fortran, PL/1, Cobol, APL and Ada.

My first choice would have been Pascal but for this application Forth appeared to be the best choice. Besides being a language, Forth forms the basis of an operating system and the Forth Interest Group[2] have made FIG Forth a public-domain product. The language is efficient, which is important when using a processor with a limited address range of 64K and it is interactive, avoiding the traps of edit-compile/load-run phases which are a left over of batch-processing systems. FIG Forth is a single-operator, single-task operating system but it has 'hooks' which allow it to be expanded into a multi-operator, multi-task system. It promotes good programming habits in that its programs are structured in blocks and work from top to bottom.

by Brian Woodroffe

The language has drawbacks – unfamiliar notation, no file structure and poor data structures – but it is readily available and has more advantages than disadvantages. The power and flexibility in Forth allows the operator to expand the language and add any desired feature, and a new version of Forth may be placed on disc by editing and compiled using the resident language to give a completely user-defined version.

Forth

Details of Forth and how it operates are available (ref. 3) and the following is a brief summary. Forth uses 16bit arithmetic and reverse Polish notation, which implies the use of a data stack. Control between executable statements, referred to as a word, is accomplished by the use of indirect-threaded code and a control stack which is separate from the data stack. Features of Forth not found in Basic are virtual memory, compiling, extensibility and vocabularies. These features make better use of the processor resources and a program written in Forth will use less memory and run faster than its Basic equivalent, often by a factor of ten or more in both cases. As Forth compiles the 'English' program into a form readable by the processor (threaded code) the operating speed will always be faster than when using Basic which stores the program as text. Memory space taken up by compiled code is much smaller than would be taken up by its equivalent in English text so larger programs are possible in a limited memory space.

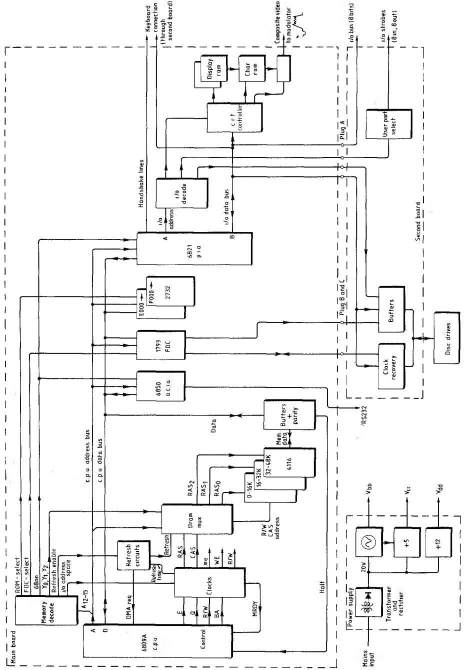
The virtual-memory feature allows the programmer to treat disc storage as processor memory so memory space is not limited by the processor but by the disc. Data is moved to and from the disc by the operating system so the programmer need not be concerned with the problem of mapping the disc memory. Vocabularies allow the programmer to keep different application programs in memory which are physically concurrent but logically separate. Further, there are features found in Forth that are not generally available in Basic such as recursion, extensibility and self-compiling. Recursion allows a portion of the code to use itself more than once at the same time and extensibility is the ability of Forth to define new control words.

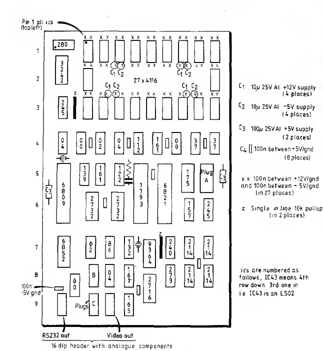
Memory choice

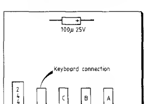
Before selecting a processor for the computer, another design decision has to be made. This concerns memory, in particular what type and how much to use. In time, memory will always become too small and too slow because of the programmer's rising expectations of what the computer should do[4], so 4116 dynamic rams were chosen because they offer the best performance in terms of cost, size and power consumption when compared with static rams such as the 2114. This decision does present some problems in that refresh circuits and three-rail supplies are required. Because dynamic rams are prone to 'soft' errors, parity-checking circuits are included in the design.

Processor choice

The Z80 microprocessor contains dynamic ram refresh circuits and CP/M is written in


Main board
Memory decode
A12–15
ROM-select
FDC-select
68nn
Y_0, Y_1, Y_2
Refresh enable
i/o address space
A
D
6809A
c p u
Control
E
Q
R/W
BA
MRDY
DMA req
Refresh circuits
Refresh
Refresh time
Clocks
RAS
CAS
me
WE
R/W
Dram mux
RAS_2
RAS_1
RAS_0
R/W
CAS
address
0–16K
16–32K
32–48K
4116
Mem data
Buffers + parity
Data
Halt
c p u address bus
c p u data bus
6850
a c i a
RS232
1793
FDC
E000
F000
2732
6821
p i a
A
B
Handshake lines
i/o address
i/o data bus
i/o decode
c r t controller
Display ram
Char rom
Composite video to modulator
Keyboard connection (through second board)
Plug A
Plug B and C
i/o bus (8 bits)
i/o strobes (8 in, 8 out)
User port select
Second board
Buffers
Clock recovery
Disc drives
Power supply
Mains input
Transformer and rectifier
20V
+5
+12
V_{bb}
V_{cc}
V_{dd}


Pin 1 all ics (top left)
Z80
27 x 4116
C1 C2
1
2
3
4
5
6
7
8
9
Plug A
B
Plugs C
100n -5V gnd
RS232 out
Video out
16 dip header with analogue components
C1 10µ 25V Al +12V supply (4 places)
C2 10µ 25V Al −5V supply (4 places)
C3 100µ 25V Al +5V supply (2 places)
C4 100n between +5V/gnd (8 places)
x x 100n between +12V/gnd and 100n between −5V/gnd (in 27 places)
z Single in line 10k pullups (in 2 places)
ics are numbered as follows, IC43 means 4th row down 3rd one in i.e. IC43 is an LS02

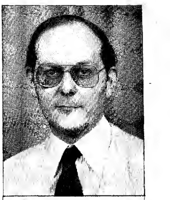

100µ 25V
Keyboard connection
C
B
A
K1160A
Pin 1
16 pin dip header with analogue components
Connect to disc drive
= 6, 100n between +5 and gnd

Complete Forth computer system has a 48K memory, floppy disc storage and memory-data parity checking but the system may be used with 16K ram and without disc storage and parity checking to reduce costs. Wire wrapping allows the computer to be built on one relatively small board with a minimum of bus buffering. A further small board holds the disc controller and i/o-port hardware. The system can be set to read most disc formats.

Z80 machine code which surely explains why it is the most widely sold processor, but to use Forth, the most suitable 8bit microprocessor is the 6809. Although most of Forth is written in Forth, the computer must execute some machine code to interpret the most primitive Forth instructions. The 6809 has indexed addressing modes (see "6809 evaluation system" by R. Coates, *Wireless World* July 1980) which suit stack operations and as said earlier, Forth uses two stacks. These examples of stack addition illustrate the merits of the 6809; they represent code of the Forth word '+' for various processors.

```
6809            Z80/8085         6800
PULU    D       POP      D       PULB
ADDD    0,U     POP      H       PULA
STD     0,U     DAD      D       TSX
                PUSH     H       ADDB   1,X
6502                             ADCA   0,X
CLC             8088             STB    1,X
LDA     0,X     POP      AX      STA    0,X
ADC     2,X     POP      BX
STA     2,X     ADD AX,BX
LDA     1,X     PUSH     AX
ADC     3,X
STA     3,X
INX
INX
```

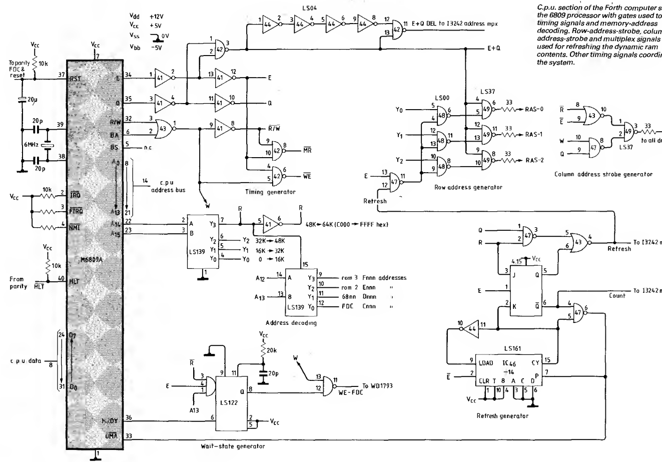
Secondly, the 6809 instruction set is particularly suited to code the crucial Forth word 'next'. The speed at which 'next' is executed determines the performance of the Forth system since this word controls the indirect-threaded code. 'Next' is called the inner (or address) interpeter to distinguish it from Forth's text interpreter which performs the function of a compiler.



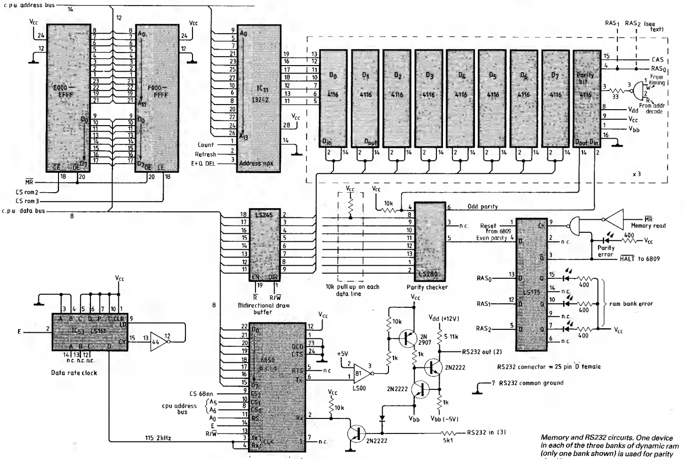
Having worked in Hewlett Packard's production and systems-engineering departments, Brian Woodroffe currently works with the company's South Queensferry research and development group and has recently been involved with designing the microprocessor control section of the HP3724/25/26A baseband analyser. Brian obtained a BA degree in engineering and economics at Downing College, Cambridge in 1970 and an MA in 1975. His computing interests include real-time control, languages and microprocessor graphics but outside electronics, his main interest — rifle shooting, in which he has represented Scotland in full bore — has been curtailed through part-time studies for an M.Sc degree in computer systems engineering at Edinburgh University

Machine code in the computer emulates Forth operation, the Y register taking on the role of the Forth program counter, and the Forth instruction-fetch cycle is a 'next' machine-code routine. So you can see that the processor choice is dominated by the speed and memory cost of the 'next' operation. Equivalent Forth 'next' operations for some microprocessors are listed below. Because the 6809 'next' operation is so short, it may be copied in line as required resulting in improved performance through avoiding the JMP NEXT instruction required for most processors.

```
6809            8088            6502
LDX 0,Y++       JMP NEXT        JMP NEXT
JMP [0,X]       LODS AX         LDY #1
(4.11)          MOV BX,AX       LDY [IP],Y
                MOV DX,BX       STA W+1
Z80/8085        INC DX          DEY
JMP NEXT        JMP WORD        LDY (IP),Y
LDAX B            PTR (BX)      STA W
INX B           (3.19)          CLC
MOV L,A         6800            LDA IP
LDAX B          JMP NEXT        ADC #2
INX B           LDX IP          STA IP
MOV H,A         INX             BCC $+4
MOV E,M         INX             INC IP+1
INX H           STX IP          JMP W−1
MOV D,M         LDX 0,X         (1.25)
XCHG            STX W
PCHL            LDX 0.X
(1.37)          JMP 0,X
                (1)
```



C.p.u. section of the Forth computer showing the 6809 processor with gates used to form timing signals and memory-address decoding. Row-address-strobe, column-address-strobe and multiplex signals are used for refreshing the dynamic ram contents. Other timing signals coordinate the system.



Memory and RS232 circuits. One device in each of the three banks of dynamic ram (only one bank shown) is used for parity checking.

Values in parentheses are merit figures obtained by multiplying the number of processor cycles by the processor cycle time then dividing by the memory-access time in the processor cycle. It is interesting to note that the 6809 fares better than the more recently introduced 8088. This is especially so when one realises that the 8088 has a 16bit arithmetic unit whereas the 6809 in common with the other processors noted has an 8bit arithmetic and logic unit (a.l.u.).

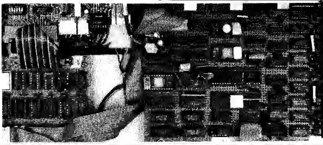
Finally, the register set of the 6809 exactly matches that which is required to operate Forth.

6809 register		**Forth operation**	
S	system stack pointer	RP	return stack pointer
U	user stack pointer	SP	data stack pointer
Y	index register	IP	instruction pointer
X	index register	W	code field pointer
D	accumulator		accumulator

Peripheral devices

Having chosen Forth and the processor to run it on, other design requirements are easily determined. These were selected to maximize the number of peripheral devices that can be easily driven. First a floppy disc was included to provide a modest amount of non-volatile memory with much faster operation than tape recorders. Mini floppy discs were chosen for two reasons, firstly because they are cheap and secondly because the data rate of eight-inch double-density drives is too high for most microprocessors to handle without direct-memory access. Further, eight-inch drives normally require phase-locked loop clock-recovery circuits and also a mains supply.

Three-inch disc drives from Sony were investigated but the data transfer rate is high so that only single-density recording could be used, which would mean wasting half of the data-storage capacity. Both these drives and eight-inch types can be used with the system, provided they run in single density. Processor memory in this system is greater than 40Kbyte so a disc capacity of greater than 400Kbyte is reasonable; one double-sided floppy-disc drive meets this requirement. It is interesting to note that the BBC Micro and Atom computer can only use single-density 5¼in disc drives because of data-rate problems.



Different types of terminal are accommodated. Operating-system words for terminals, KEY, TERMINAL and EMIT, are vectored so that they may be changed on-line between terminal types. At switch-on the system automatically sets vectors for the available terminals. These terminals are either serial RS232 or 8bit parallel for a keyboard such as the RCA VP601/611 and integral video compatible with 625-line tv, displaying 1,024 characters in 16 lines (the EF96364B controller may be used for 525 lines). The video section has its own memory, leaving 48K of memory free for other programs. Bit-mapped graphics video is best handled through a secondary processor connected to the user ports. A number of definable i/o ports are spare to allow for expansion of the system. Certain design features were included to reduce cost. By keeping the computer system down to one board, bus drivers necessary to overcome capacitance encountered in larger systems are avoided. Another reason for avoiding these buffers is that they cause delays which eat into the access time available from communicating devices. The switch-mode power supply used means that a readily obtainable transformer with a single secondary winding may be used to provide all three rails (+12, +5 and −5V). WW

Next article describes computer circuitry.

References

1. E. W. Dijkstra, How do we tell truths that might hurt? ACM Sigplan notices, vol. 17 no 5, May 1982, p.14.
2. Forth Interest Group (FIG), PO Box 1105, San Carlos, CA94070, USA.
3. L. Brodie, Starting Forth. Prentice Hall, 1981
B. Allan, Forth: a threaded interpretive language, *Wireless World* Nov. 1982 p.74.
G. Feierbach, Forth – the language of machine independance, Computer Design, June 1981 pp.117-121.
Byte, August 1980 for various Forth articles.
4. W. Wulf, Compilers and Computer Architecture, *Computer*, July 1981 p.41.

continued from page 50

Z8 Besic listing for eight-channel a to d converter (@ ≡ byte, % ≡ hexadecimal)

```
   1 PRINT "SILICONIX LD120/121A TO Z8
     INTERFACE"
   2 PRINT "HIT ANY KEY TO RUN";: GO
     @%61,%07:R=USR(%54)
   5 A =  (        )  :  B =  (        )
  10 C =  (        )  :  D =  (        )
  15 E =  (        )  :  F =  (        )
  20 G =  (  see*  )  :  H =  (  see*  )
  25 I =  (        )  :  J =  (        )
  30 K =  (        )  :  L =  (        )
  35 M =  (        )  :  N =  (        )
  40 O =  (        )  :  P =  (        )
  45 N  =1  : Z=%80
  50 @2=Z
  55 @3=%A0:03=00
  60 @2=%00
  70 W  =W+1:IFW<15 THEN 70
  80 W  =0:GO@%1400 (m.c. routine for
     collection of data)
  85 IF N=1 THEN X=A: Y=B: GO TO 130
  90 IF N=2 THEN X=C: Y=D: GO TO 130
  95 IF N=3 THEN X=E: Y=F: GO TO 130
 100 IF N=4 THEN X=G: Y=H: GO TO 130
 105 IF N=5 THEN X=I:  Y=J: GO TO 130
 110 IF N=6 THEN X=K:  Y=L: GO TO 130
 115 IF N=7 THEN X=M: Y=N: GO TO 130
 120 IF N=8 THEN X=0:  Y=P: GO TO 130
 125 GO TO 45
 130 Z=Z+%10
 135 V=@%26+@%25×10+@%24×100
     +@%23×1000+@%22×10000
 140 IF V>X THEN PRINT "CHANNEL";
     N;"OVERRANGE";:GO TO 1000
 145 IF V<Y THEN PRINT "CHANNEL";
     N;"UNDERRANGE";:GO TO 1000
 150 N=N+1: GO TO 50
1000 GO@%61,%07:PRINT @%22;".";
     @%24;@%25;@%26;
1010 PRINT"MAX";X;"MIN";Y
1020 N=N+1 : GOTO50
```

*Limits entered here for process monitoring.

Machine code routine

Line	Assembler	Hex
200	LD % F7, # % 41	E6 F7 41
210	LD % F6, # % 0F	E6 F7 0F
220	AND 3, # % 04	56 03 04
230	JRZ, * 220	6B FB
240	LD % 22, 2	E4 02 22
250	AND 3 # % 04	56 03 04
260	JR N2, * 250	EB FB
270	CLR % 21	B0 21
280	AND 3 # % 08	56 03 08
290	JRZ, * 270	6B FB
300	PUSH 2	70 02
310	INC % 21	20 21
320	CP % 21, # 4	A6 21 04
330	JRZ 2, * 350	6B 07
340	AND 3, # 08	56 03 08
350	JR N2 * 320	EB FB
360	JR * 270	8B EB
370	POP % 26	50 26
380	POP % 25	50 25
390	POP % 24	50 24
400	POP % 23	50 23
410	RET	AF

Using the Z8 microprocessor

The potential of the Zilog Z8 microcomputer for low-volume or one-off applications has not been well publicised. Introduced in 1981, the chip includes all the blocks that you would find in a standard computer system including c.p.u., ram, i/o and a rom containing Tiny Basic. In addition there are two timers, one for serial i/o, the other available to the user. External memory of 60K can be added for program storage together with 60K of data storage. The evaluation kit used in this was supplied by Comart International and possesses a comprehensive monitor enabling machine-code routines to be written in Assembler and stored in eprom. The Tiny Basic name does not reveal the full potential of itself, and with an 8MHz clock driving the Z8, it executes most of the instructions in 1.5 to 2.5μs.

Forth computer

In describing memory and i/o interface circuits surrounding the 6809 microprocessor, Brian Woodroffe introduces more features of his FIG Forth computer in this second article.

by Brian Woodroffe

The system may be used in partial form. Operating-system and language software exist in eprom, so the computer will work without a floppy-disc drive. Many computers use eprom as a bootstrap to load an operating system from disc, making a disc drive mandatory. Although omitting the disc drive reduces cost by almost half, virtual-memory features of Forth are lost, resulting in a significant degradation of performance. Fewer than one third of the memory devices are essential. Parity-error checking may be omitted. When the system is turned on, it only demands 16K of ram and as more is added the memory map is changed on line, Table 1 (see over).

Circuit description

Memory. Eproms containing fixed instructions of the Forth machine and M6809 peripherals pose few problems. These devices occupy the top 16K memory locations because the 6809 reset vector is in this area and decoding is simple using a dual two-to-four-line demultiplexer i.c. (LS139). Dynamic ram occupies the remaining 48K addresses from 0000 to BFFF. Logic i.cs used to glue the main items together are low-power Schottky devices, chosen for their speed and low power consumption. Standard t.t.l. parts could be used, except in the timing chain for the dynamic rams and on the microprocessor memory and address buses; nmos microprocessor parts have very low driving capability and low-power Schottky inputs require less current than standard t.t.l.

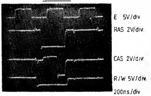
Dynamic rams consist of an X-Y matrix of capacitor storage cells. Access to a bit (storage cell) is gained by first addressing the matrix row. This address is clocked in by the falling edge of the row-address strobe (RAS) and data from all 128 cells in the row are transferred to row buffers. When the column-address strobe (CAS) is true, i.e. low, the column address on the address pins selects one of the row buffers, causing its data to be passed to the output pin. Timing constraints on these actions are fortunately not stringent relative to the time available in a processor cycle

Multiplexing of the 14 address lines onto the seven address pins is done with an I3242 multiplexer. In this design, writing is carried out by the early-write cycle. Within the early-write cycle the write signal is made true before the column-address strobe acts. When CAS becomes true, data on the data input overwrites that of the selected row buffer. Then when the address strobes become false, data from the row buffers are returned to their respective cells, so writing the input data into the X-Y matrix.

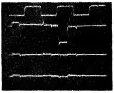
Two clocks, E and Q, divide the 6809 processor cycle into four parts. The first quarter of the cycle is used to precharge the rams and as dynamic rams consume most power when the row-address strobe is applied, the selected bank of rams only receives this strobe on the rising edge of clock Q. The address multiplexer is then switched by a delayed Q-clock edge to apply column addresses, leaving sufficeint settling time before the E-clock acts.

During a reading cycle the column-address strobe is made true half way through a cycle (rising edge of E Clock) so that data may be made available by the RAS-selected rams, through the LS245 buffer, to the M6809 before its set-up time. All of the rams receive CAS but only those receiving RAS pass data to the bus.

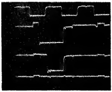
Write cycle



Refresh cycle



Read cycle



During a writing cycle data is not made available by the M6809 until the second half of the cycle so CAS is delayed until the falling edge of the Q signal.

Refresh generator

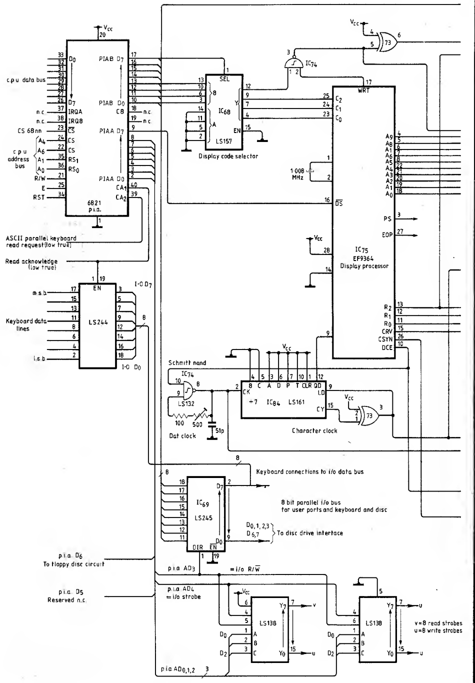
Storage cells in dynamic rams, being capacitors, lose their charge so they must be 'refreshed'. Any memory action refreshes the selected row through data being read into the refresh buffer and returned at the end of the cycle. Unfortunately, program flow will not normally refresh all the ram rows in the allotted time of 2ms and a refresh generator is required.

There are three ways of refreshing rams. In burst refresh, normal processor action is suspended and the refresh generator cycles through all 128 rows (for a 16K ram) and returns control to the processor for the remainder of the 2ms. This results in the processor stopping for 128 memory cycles (85μs at the clock speed used). Such a time lapse is unacceptable in this application for the disc drive can require communication with the microprocessor once every 32μs during sector read/write operations.

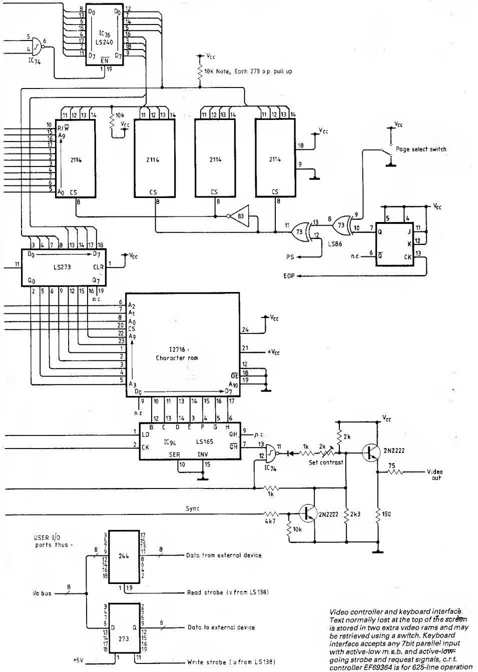
So, distributed refreshing is required, that is, each successive row is refreshed at 14μs intervals. Distributed refresh generators demand that the processor does not have access to memory while the row is refreshed. The processor may be stopped for this period but a more efficient method is to use a circuit that recognizes when the processor is not using memory and performs what is called a distributed hidden-refresh cycle. This method was chosen.

The refresh generator divides time into 14 cycle quantums using an LS163 counter and generates a refresh-request signal once each period (14μs × 128 cycles = 1.7ms). By monitoring address lines $A_{14,15}$ during the first quarter cycle, the generator knows when the processor does not require access to memory. Having recognized this it generates a refresh-request signal and the I3242 multiplexer places the refresh address on the ram address lines and all row-address signals are set true for a quarter of a processor cycle. During the refresh cycle the column-address strobe is false to inhibit the rams. The address multiplexer advances for the next address and the generator does not demand further refreshes since a flip-flop is set.

It is unlikely that the M6809 will make 14 consecutive memory cycles since all instructions except NOP, SEX and DAA provide non-memory cycles. Should this happen, the refresh flip-flop being reset



Vcc
6821
p.i.a.
c.p.u. data bus
D0
D7
PIAB D7
PIAB D0
PIAA D7
PIAA D0
IRQA
IRQB
CS 68nn
c.p.u. address bus
A4
A6
A1
A0
R/W
E
RST
RS1
RS0
CA1
CA2
CB
n.c.
ASCII parallel keyboard read request (low true)
Read acknowledge (low true)
SEL
IC68
LS157
EN
Display code selector
IC74
IC75
EF9364
Display processor
WRT
1.008 MHz
DS
PS
EOP
R2
R1
R0
CRV
CSYN
DCE
73
LS244
m.s.b.
Keyboard data lines
l.s.b.
I-O D7
I-O D0
Schmitt nand
LS132
100
500
51p
Dot clock
IC84
LS161
÷7
CK
CLR
QD
LD
CY
Character clock
Keyboard connections to i/o data bus
IC69
LS245
DIR
8 bit parallel i/o bus for user ports and keyboard and disc
D0,1,2,3 D6,7
To disc drive interface
p.i.a. D6
To floppy disc circuit
p.i.a. D5
Reserved n.c.
p.i.a. AD3
= i/o R/W
p.i.a. AD4
= i/o strobe
LS138
Y7
Y0
v
u
v=8 read strobes
u=8 write strobes
p.i.a. AD0,1,2



Video controller and keyboard interface. Text normally lost at the top of the screen is stored in two extra video rams and may be retrieved using a switch. Keyboard interface accepts any 7bit parellel input with active-low m.s.b. and active-low-going strobe and request signals, c.r.t. controller EF69364 is for 625-line operation but a version is available for 525-line tv.

Table 1. Example of how the memory map may be changed when more than 16K of ram is used.

```
FORTH HEX
SMAX DUP @ 4000 + SWAP !              ( allow more data stack)
S0 DUP @ 4000 + SWAP !                ( move data stack)
SPI SMAX DUP @ 4000 – SWAP !          ( reset data stack)
R0 DUP @ 4000 + SWAP !                ( move return stack)
TIB DUP @ 4000 + SWAP !               ( move terminal input buffer)
' FIRST @ DUP @ 4000 + SWAP !         ( move Forth virtual memory buffers)
' LIMIT @ DUP @ 4000 + SWAP !         ( IE 'FIRST' and 'LIMIT')
FIRST DUP PREV ! USE !                ( point virt. memory pointers to virt. memory)
DPMAX DUP @ 4000 + SWAP !             ( move limit of dictionary up)
DECIMAL                               ( return to decimal arithmetic)
```

and the counter carry being set (refresh quantum finished), processor action is suspended by a dummy direct-memory-access cycle which guarantees a non-memory-access cycle.

Parity checking

Capacitance used to store data in dynamic rams is so small that naturally occurring charged particles (alpha particles) have a charge great enough to corrupt data should they hit a cell. Improved coatings on dynamic-ram dies have reduced this effect to give an error rate below 0.1%/1000h for 16K dynamic memories[5]. It is impractical to include error correction in small 8bit memories but parity checking to halt the processor when an error occurs is not.

An odd-parity bit, generated by an LS280 parity checker when a byte is written into memory, is stored with the other eight bits. During the write-cycle the parity-ram data output is in its high-impedance state and the floating EO input is high. The parity device output is clocked into the ram input and correct parity is looked for when memory is read. On reading, the data output drives the parity checker and the error signal is passed to the error latch with the row-address strobe signals. If an error exists, the RAS line concerned is latched, a led indicates which memory bank contains the error, and the processor halts.

Memory speed and drive

Input characteristics of dynamic ram are quite different from those of t.t.l. Ram inputs are capacitive, which especially affects signals common to many inputs like RAS, CAS and WE, and they require little direct current. When driven directly from low-power Schottky t.t.l. these inputs can cause considerable overshoot that can result in exceeding device specifications and longer access times through the time taken for the voltages to level out.

To reduce ringing, some form af matching is required. Series matching is most appropriate since it does not increase static loading. The ideal driver would produce a slightly under-damped response but because t.t.l. drive characteristics are asymmetric a compromise had to be made in the resistance value. Control signals are driven from LS37 clock drivers to ensure adequate drive toward the 5V rail. Resistance values are not critical for this relatively slow memory and the original even worked faultlessly with no damping resistors and standard LS00 drive.

On analysing the timing requirement of the ram/M6809 interface I noticed that the most readily available 200ns rams leave a lot of spare time – so much so that these devices could theoretically be run with a 666ns cycle time instead of the standard 1µs. This was, of course, tried. Not only was it tried with the faster M6809A processor but also with the standard device. In both cases functioning was faultless. This is not to say that all 1MHz parts will run at higher speeds but certainly 200ns access time rams will work at 1.5MHz. So for the cost of a new crystal the through-put of the system was improved by 50%.

Peripherals

To ensure that 1MHz peripheral devices such as the 6821 peripheral-interface adapter and the 6850 communication-interface adapter operate correctly, the memory-ready signal (MRDY) is used. Whenever peripherals are addressed MRDY is held false by an LS122 monostable multivibrator which extends the memory-access time. An M6850 communication device forms the RS232 interface and the clock frequency for it is crystal derived. Currently the 1.5MHz c.p.u. clock only allows 1800bit/s and an external baud generator is an attractive proposition. Both –5 and +12V supplies are used for the RS232 interface. Current from the –5V supply is so low that the RS232 driver has an active current limiter; the +12V drive is resistive.

Many of you will not have an RS232 terminal and will wish to use a separate keyboard and domestic tv. The keyboard interface will accept any 7bit parallel input signal with active-low most-significant-bit and active-low-going strobe and request signals. Two spare hand-shake lines on the p.i.a. and an output port could form a Centronics-type printer port.

An EF69364A video i.c. provides timing signals necessary for a 625-line tv; a 96364B device will provide signals timed for 525-line tv. Control code for the video i.c. is supplied through an LS157 quad two-to-one-line multiplexer and for normal display characters (p.i.a. B D_7=0) a fixed control code is set. When control characters (hexadecimal 0 to F) are used the p.i.a. supplies the relevant code through the multiplexer (p.i.a. B D_7=1) to the EF69364. As the c.r.t. gun scans the screen, the EF69364 selects the character to be displayed from the display ram and latches it into an LS273.

The video i.c. was designed for use with ram that has separate data input and output lines (2101 ram) so the circuit was modified to allow 2114 rams with common i/o to be used. Character-code from LS273 and row information from 69364 is supplied as an address to a character rom (a specially programmed 2716 eprom). Each character position is allocated a 7-wide-by-12-high character block.

Referring to last month's article, the signal name at pin 6 of IC_{41} is active low and should read $\overline{R}$, as should the signal name at the junction of IC_{47} pin 2 and IC_{45} pin 3. On page 57, pins 13, 12 and 5 of the LS175 should be labelled Y_0, Y_1 and Y_2 respectively.

A set of three programmed roms is available from Brian Woodroffe at 632 Queensferry Road, Edinburgh for £23.50 inclusive. Technomatic (see advertisers' index) will supply all i.cs mentioned in this article.

Disc-drive interfacing is described in the next article.

References

5. E. Westfield, Memory system strategies for soft and hard errors, Wescon '79.

continued from page 48

indebted to Keith Frewin, who wrote the SOFTBOX software, for providing roms 385 and 386.

References

1. P. G. Barker, Data Transmission Between Micros, *Electronics and Computing Monthly*, 2(5), 1982, 21-25 and 46-49.
2. P. G. Barker, Introducing CP M, *Electronics and Computing Monthly*, 1982, in press.
3. A. Osborne and J. Kane, An Introduction to Microcomputers: Volume 2 – *Some Real Microprocessors*, Osborne & Associates Inc, California, 1978, Chapter 10, pp29-49.
4. Commodore Business Machines Ltd., CBM PET 3032N Professional Computer User's Manual, Publ. No. 320856-3, June 1979.
5. Commodore Business Machines Ltd., CBM PET Series 8000 User's Guide, Publ. No. 320894, 1981.
6. MOS Technology Inc., MCS6500 Microcomputer Family Hardware Manual, Publ. No. 6500-10A, 2nd Edition, January 1976.
7. E. Fisher and C. W. Jensen, PET and the IEEE-488 Bus [GPIB], Osborne/McGraw-Hill, California, 1980.
8. National Semiconductor Corporation, *SC/MP* Technical Description, Publ. No. 4200079B, September 1976.
9. I. Williamson and R. Dale, Understanding Microprocessors with the Mk 14, The Macmillan Press, Bristol, 1980.
10. Small Systems Engineering Ltd., 2-4 Canfield Place, London NW6 3BT, UK, SOFTBOX User Manual, Revision 3, 1981.
11. J. Kane and A. Osborne, An Introduction to Microcomputers: Volume 3 – Some Real Support Devices, Osborne & Associates Inc., California, 1978.
12. P. G. Barker, Computers in Analytical Chemistry, Pergamon Press, Oxford, 1982, in press.
13. MOS Technology Inc., KIM-1 Microcomputer Module User Manual, Publ. No. 6500-15B, 2nd Edition, August 1976.

Forth computer

Interface circuits and software for disc-drive control are main subjects of Brian Woodroffe's third article describing his 6809-based microcomputer. First, operation of the video controller is concluded and i/o software discussed.

by Brian Woodroffe

Character-code and row information for the video-controller i.c. is supplied as an address to a character rom. Character information for each row is fed to an LS165 shift register and serial output from this register is combined with synchronization signals in an analogue gate to give a standard 1V p-p composite-video signal which is subsequently fed to a u.h.f. modulator.

The dot clock, consisting of a Schmitt-trigger relaxation oscillator, should be adjusted to the minimum frequency to minimize the luminance bandwidth required in the monitor consistent with all text displayed on the screen. Character values 10 to 1F hexadecimal are programmed into the character rom to give coarse graphics. Two 2114 rams hold enough information for one 1024-character Forth screen to be displayed.

Two further video rams store text normally lost at the top of the screen. A switch allows a page of lost text to be displayed.

Terminal and i/o software

The Forth reset routine checks to see if there is an M6850 present and if not automatically redirects terminal i/o routines from the RS232 interface to the p.i.a. for parallel i/o. Forth words giving access to user ports are included in this operating system. These words, P@and P! act in the same way as Forth words @ and ! except that they allow access to user i/o ports.

The software-driven output word, P!, makes data available on the p.i.a. B lines then activates the address coded on the A lines. On-input, P@, reads data while the port address is made. Output ports ideally connect to LS273 latches and input ports to LS244 buffers. Port-strobe lines are decoded from the p.i.a. A lines using LS138 three-to-eight-line decoders. Eight read and eight write ports can be connected to this hardware and if more ports are needed then a further 6821 p.i.a. could be connected and mapped into the USER variable-address area. Cursor control codes, i.e. decimal codes for EMIT, are as follows.

8	left (backspace)
9	right (tab)
10	down (line feed)
11	up
12	home and erase
13	carriage return
14	home
15	carriage return and line erase

Disc interface hardware

Interfacing to the floppy disc[6] is done using the most readily available controller since it is cheaper than using s.s.i./m.s.i. devices. Complexity of the WD1793 controller is comparable to that of the 6809. The first problem was interfacing an 8080 style peripheral to the M6809 bus, the main difficulty being the writing data-hold times.

The problem of data-hold times was solved using the memory-ready signal, MRDY, which when active (low) holds the processor clock cycles in an E-not-Q state for at least one quarter of a bus cycle. This quarter cycle provides the hold time. The memory-ready signal triggers a monostable multivibrator each time the procesor wants access to peripheral-drive address space between C000 and DFFF on the rising edge of the Q clock and this signal forms the floppy-disc controller write signal.

A read signal is derived from clocks E and Q. Interrupt and data-request outputs of the floppy-disc controller are connected to the processor FIRQ pin so that data transfer can take place using the M6809 SYNC instruction. As noted before, a floppy-disc drive's data rate can cause problems when d.m.a. is not used. In double-density recording on a 5¼in floppy using a WD1793 controller, the worst-case data-transfer rate is 27μs/byte. Coding is shown in Table 1.

The trick is that SYNC stops the M6809's execution without affecting the clocks until the floppy-disc controller interrupt occurs and the processor resumes execution. This provides quick synchronization between the processor and controller. Despite that modifying the direct-page register gives quicker access to the f.d.c. which is in high memory, this feature was not used because of the extra coding needed. Had the processor clock been slower this alternative might have been necessary.

Interfacing the floppy-disc controller to the drive is the next problem. Most of this is covered in an ANSI standard[7] but the problem of clock recovery remains. Because of mechanical constraints, data read from disc will not be synchronous with any processor clock so clock information contained in the data stream must be extracted. In single-density recordings each bit cell has a clock bit and a possible data bit (no data bit is zero) and in double-density recording the position of the bit within the cell determines whether it is a one or a zero. A clock synchronous with incoming data is required to determine the incoming bit's position.

Although it gives the best performance, a phase-locked loop circuit was rejected on grounds of cost. Instead a crystal clock running at eight times the nominal read clock is used and a divide-by-eight version of this clock is phased with the incoming data to recover the original clock. First the incoming bit stream is synchronized to the crystal clock (×8) to produce pulses with accurately defined widths using an LS74. This pulse stream is fed to the floppy-disc controller (RAW READ).

The reading clock is provided by an LS161 counter which is normally held off until the controller wants to read the disc, when the counter is enabled by the read-gate signal. This counter would normally free run at about the nominal clock rate, but it is synchronized by applying the raw read signal to its load input. The load frequency locks its D output (READ CLOCK) so that it changes mid-way between input bits. As the maximum number of bit cells without read bits is three, the recovered clock never gets too far out of phase.

Table 1. Code showing how the M6809 SYNC instruction is used for floppy-disc drive data transfer.

```
         STB FDC      F7C000            send command byte to f.d.c
BRED2    SYNC         13           2    wait for f.d.c. response
         LDB FDC      F6C000       5    get status
         BITB #2      C502         2    test byte-in
         BEQ BRERR    2710         3    no, then error
         LDA FDC+3    B6C003       4    get byte
         STA 0, Y+    A7A0         7    store, advance pointer
         LEAX -1,X    301F         5    reduce count
         BNE BRED2    26EF         3    loop back
BRED3    LDB FDC      D600              wait till
         BITB #1      C501              f.d.c. finishes
         BNE BRED3    26FA
BRERR    RTS          39
```

32 cycles at 1.5MHz ≈ 22μs

Upon entry B=command code
Y=pointer to data destination
X=byte counter

The author is with Hewlett Packard in research and development.

Problems with phasing are most noticeable when double-density recording is used, so a means of preventing bunching of the bits is used. Precompensation[8] prevents bunching by moving the written data bits slightly relative to the nominal position in a bit cell so that when the data is read back the bits appear to be in their correct positions. The matter of precompensation depends on the drive used. For those drives that do not require precompensation, including the TEAC FD50A used in the original design, the precompensation circuit is omitted.

The disc should be set to respond to its address and head-load on drive select and not to the motor-on signal, i.e. the TEAC FD50 disc drive should be set as follows (for further drives, follow the same pattern).

DRIVE 0
HS=set, MX=set, DS0 set,
DS1,DS2,DS3=unset,
HM=disconnected.
DRIVE 1 (if fitted)
HS=set, MX=set, DS1=set,
DS0,DS2,DS3=unset,
HM=disconnected.

Disc-interface software

Under command of the c.p.u., the floppy-disc controller takes care of head positioning, sector positioning, data serialization and cyclic-redundancy checking. As soft sectoring is used, sector positioning is determined by the address record read from the formatted disc. The controller may be programmed to format the disc. So long as certain inter-record gap and record sizes are adhered to, the formatted disc capacity may be increased, Table 2.

Different systems use different sector formats[9], numbers and sizes of sectors and sector numbering systems. In this system, all variables associated with disc formatting are defined by the user which means that most disc formats may be read. The sector size is written into the address record of each sector so it is possible for the system to adjust its buffer size to that of the disc. Forth word ?DISC is included to read the current disc and set parameters termed DENSITY, B/BUF and SEC/TRK to those associated with the disc. Only formats mentioned in Table 3 apply to the disc format program and ?DISC.

When formatting a disc, it can be advantageous to interleave the sectors on a track. With this in mind a dummy word SKEW was included which is currently defined as no operation, but it may be redefined to perform an interleaving algorithm during formatting, Table 3. Defining Forth word FORMAT for disc formatting is shown in Table 4.

Forth treats all disc memory systems in the same way, i.e. as a contiguous set of 1024byte screens, hence the choice of a v.d.u. Main Forth words used to gain access to screens on a disc are R/W, which moves data between a disc and memory, and BLOCK. As disc sector size depends on format, words BLOCK and constants B/BUF, bytes-per-sector, SEC/TRK, sectors-per-track, TRK/SIDE, tracks-per-side and SIDE/DISC provide a means for Forth to work out which sectors make up a screen. The size of virtual memory buffers in Forth should be the same size as a sector.

Table 2. Capacity of a formatted disc may be increased provided that certain record sizes and gaps are not exceeded.

Density	Single		Double	
Bytes/sector	128	256	256	512
Sectors/track	16	10	16	10
Bytes/track	2048	2560	4096	5120
Bytes/disc	82K	102K	160K	205K
Relative	100%	125%	200%	250%

Time taken for the head to position itself over the relevant track is a major constraint when using disc drives. Other time factors for a 5¼in floppy-disc drive are motor start-up time, head-load time and rotational latency. To speed up access time for double-sided discs it is usual to physically combine two tracks on opposite sides of the disc into one logical track. This minimizes head seek time for it is likely that the sector required will be on the same bigger logical track and the time taken to gain access to the other side of the disc is governed by the time taken for an electrical switch to act rather than by the delay of a mechanical head seek. But since Forth treats all discs in the same way, including this feature would have meant that one could not mix single and double-sided discs.

When using the Teac FD50A disc drive, access time is dominated by the start-up time of 1s. If faster disc drives are used, time constants may be changed (discussed in a following article). Start-up time and head-stepping rate constants are moved into ram from eprom by the Forth start-up word COLD and may be modified to suit faster drives. Forth constants normally hold the values of constants in the parameter-field address (p.f.a.) but as this system is rom based, modification of the constants would not be possible so they are coded with a new routine which stores the value in ram. This list shows how the constant DENSITY is altered from single to double density and gives other constants and their meanings.

DENSITY =1 (double density, 0 for single density)
B/BUF =512 (number of bytes per disc sector)
SEC/TRK =16 (number of sectors per disc track)
TRK/SIDE= (number of tracks on disc, normally 35-40 for a mini-floppy)
SIDE/DISC=1 (2 for double-sided)
SEC-OFST =1 (for numbering sectors 1 to n, 0 for numbering 0 to n--1)

```
1          (value to store, returned after
            execution of DENSITY)
' DENSITY  (find DENSITY p.f.a.
            address)
@          (p f.a. in this special constant
            points to constant position)
!          (store 1 there)
```

Power supply

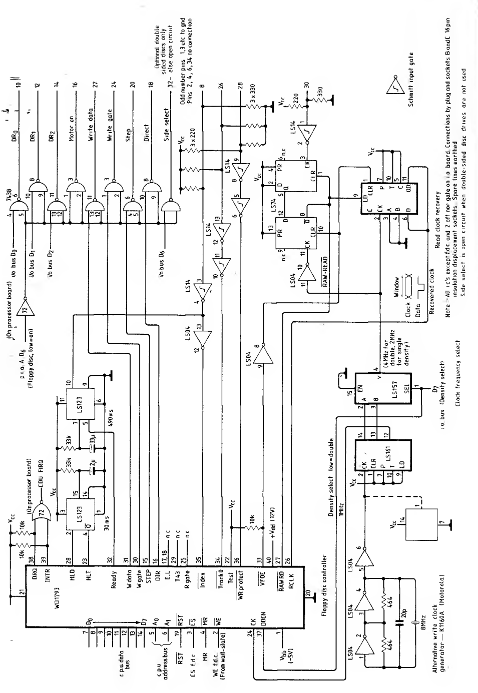
Only one 15V secondary winding is required on the transformer to provide a low-current −5V supply for biasing the dynamic rams, +12V for the rams and floppy-disc drive and +5V for all logic circuits. A minimum value for the unregulated supply is determined by the 12V rail; unregulated input should be 20V to ensure adequate regulation with low mains supplies. Heaviest current demands are on the 5V supply and using a linear regulator to provide this rail would have resulted in excessive heat generation with a loss of efficiency so a switching regulator was designed.

Table 3. Example of a routine for defining dummy word SKEW to give interleaved formatting.

```
FORTH HEX                ( select Forth and hexadecimal number base)
: SKEW1 DUP              ( new word, duplicate sector # to be interleaved)
 1 AND IF                ( only even sectors are interleaved)
  SEC/TRK 2 / FE AND     ( sector offset by half the disc)
  + SEC/TRK MOD          ( add offset and keep with 0 .. , n−1 sectors on
                         track)
  THEN ;
' SKEW1 2 –              ( find c.f.a. of new interleaving address)
' SKEW !                 ( find old skew p.f a. and overwrite no-op there)
```

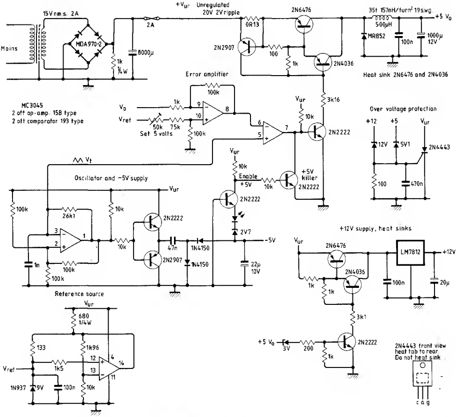
Table 4. Routine for defining Forth word FORMAT for disc formatting.

```
: FORMAT                          ( start compiling the word format)
 0 DR-SEL 100MS RATE CMND         ( turn disc drive on, seek track 0)
 #SIDES 0 DD                      ( do for both sides)
  TRK/DISC 0 DO                   ( do for all tracks)
   DP @                           ( save pointer to scratch area)
   I J BLD-TRK WR-TRK             ( build up image of track, write it out)
   ." track/side/status=" I . J . . CR   ( inform user, 0=good status)
   1 STEP                         ( step in for next track)
   DP !                           ( recover scratch area)
   LOOP
  RATE CMND LOOP                  ( for other side)
 DE-SEL ;                         ( turn drive off, finish compilation)
FORMAT                            ( carry out format)
```



Note - All i c's except f d c and 2 off nor gate on i o board. Connections by plug and sockets B and C 16pin insulation displacement sockets. Spare lines earthed

Side select is open circuit when double-sided disc drives are not used



Switch-mode power supply uses one 15V r.m.s. secondary winding for +12V, −5V and high-current +5V rails. Frequency of the relaxation oscillator is 17kHz, giving the best compromise between smoothing component sizes and loss in efficiency due to switch transition times eating away at the duty cycle. Gating ensures that dynamic rams receive their three supply rails in the correct sequence and s.c.rs provide overvoltage protection.

After bridge rectification and capacitive filtering, the 15V r.m.s. transformer output gives approximately 20V. Dynamic rams are sensitive to the sequence in which power is applied to them so the supply had to be designed so that −5V appears first, followed by +5V then +12V.

Heart of the switch-mode power supply is a relaxation oscillator, the squarewave output of which feeds a charge pump to produce about −20V peak. This is regulated by a zener diode to produce −5V. Reference for the +5V supply is a 10V zener diode connected in a feedback loop to maintain constant current even when the 20V unregulated supply varies. An error signal derived from the +10V reference and +5V supply, and the relaxation oscillator triangle wave are fed to a comparator. A portion of the triangle wave depending on the magnitude of the error signal is fed to the switching transistor. This pulse-width modulated base drive is disabled when the −5V supply is not present.

The free-wheel diode, inductor and smoothing capacitor are fed by the switching transistor and are chosen with the operating frequency in mind. Around 17kHz is used since it is the best compromise between high-frequency losses and component size. At low frequencies the smoothing capacitor and choke become too large and at high frequencies the switching transition time takes up a large portion of the cycle time and efficiency is reduced.

Unregulated supply passes to the 12V monolithic regulator under control of a transistor switched by the +5V supply. To prevent overvoltage problems, an s.c.r. is included which switches on and blows the secondary winding fuse if either the +5 or +12 rails rise too high.

Disc interface uses a readily available controller which works out cheaper than an equivalent circuit using s.s.i./m.s.i. devices. Clock information in data read from disc is synchronized using a crystal-controlled oscillator running at eight times the rate of the incoming-data clock. The prototype computer has a standard Teac 51/4in floppy-disc drive.

To be continued with construction tips, parts list and vocabulary.

References

6. J. R. Watkinson, Disc drives, *Wireless World*, Mar.–May & July-Dec. 1982, Jan.-Mar. 1983, especially Oct. & Nov. 1982.

7. American National Standard, Interfaces between flexible disc drives and their host controllers, X3.80-1981 (ANSI).

8. J. F. Hoeppner & L. H. Wall, Encoding ecoding techniques double floppy-disc capacity, *Computer Design*, Feb. 1980, pp 127-135.

9. E. Kadison, 5.25in floppy-disc formats, *Electronic Design*, 23 Dec. 1982, p. 135

Forth computer

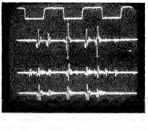
Construction tips for the 6809-based Forth computer – part four.

by B. Woodroffe

Most of the prototype version of this computer was constructed on one wire-wrap board. The number of signal buses rendered anything other than a multilayer printed circuit board an impractical solution without splitting the circuit into sections. Splitting the circuit was rejected to eliminate buffers associated with long cable runs. Wire wrapping provides connections at least as good as solder joints through cold welding between the wire and edges of the pin.

All main memory, refresh circuit, microprocessor rom and interface i.cs are mounted on the main 229 by 178mm board, as are the video-display processor and memory. The analogue video gate and RS232 driver are built on two 16-pin dip headers. User-port hardware and the disc-drive interface between the floppy-disc controller and the drive are housed on a second wire-wrap board. There are many connections on the board so a powered wrapping tool, a stripping tool and different coloured wires for different functions are useful. Copper-clad board was used for the power supply, which should be constructed before the main processor board.

Dynamic ram takes little static current but substantial pulses, reaching toward 80mA per device over a few nanoseconds on some clock edges. Although the rams work within a 10% voltage tolerance, for reliable operation substantial local decoupling must be included in the +12 and −5V rails to overcome power-line inductance; each ram has a 0.1μF ceramic capacitor on both supplies. Further 10μF bulk decoupling capacitors were used, one between each four devices. Decoupling capacitors for the 5V rail were used throughout the design at the rate of one 100nF component for each six i.cs. As with the RAS/CAS/WE damping resistors, the design seems robust since the ram was initially built and worked without decoupling (see photograph).



Voltage transients at the 4116 dynamic rams showing from top to bottom the E clock signal and +12V, +5V and −5V supply lines with a 200ns/div timebase

This is a large project and all construction errors were found to be the result of either miswiring or plugging in the i.c.s wrongly. Dynamic rams I currently use got very hot when I plugged them in back-to-front. Construction should start with a minimum system, i.e. c.p.u., p.i.a., eproms and a 16K ram. At switch on, the lamp connected to the p.i.a. B-port D_0 line will go on then off. The state of this lamp then monitors the state of i/o data on the line. Ram-select lamps will stay off. V.d.u. hardware is self-contained so an idea of its performance can be seen on a tv screen without involving the main processor as the video i.c. generates its own characters.

Connection of the parity circuit to HALT should only be made after the ram circuits are known to work, i.e. when the system ready message can be displayed consistently. Should the RS232 connection fail to work, the most likely cause, especially if a signal at the a.c.i.a. output can be seen on resetting, is that data lines on pins two and three are crossed. Another problem could be that the RS232 terminal

Brian Woodroffe works in research and development at Hewlett Packard.

Main-board components

Resistors

Value	Qty	Function
10k	8	pull-up, FIRQ, IRQ, NMI, VFOE, RESET, video and RS232 output
10k	2	pull-out parity, video ram, 9-resistor sil packs
100	1	dot-clock
500	1	dot-clock trimmer
20k	1	monostable timing, 5%
400	4	pull-up, led
33	5	damping, RAS, CAS, R/W
75	1	video output
150	1	video output
1k	5	video and RS232 output
2.3k	1	video output
4.7k	1	video output
2k	1	video output
2k	1	video output, trimmer
5.1k	2	RS232 output

Capacitors

Value	Qty	Function
100μ	2	+5V decoupling, 25V
20μ	2	+12V decoupling and reset, 25V
10μ	8	−5V and +12V decoupling, 25V
100n	57	−5, +5 and +12V decoupling
20p	2	crystal decoupling, 10%
51p	1	dot clock, 5%
20p	1	monostable timing, 5%

Integrated circuits

Ref	Qty	Pins	Type	Comments
11	1	14	LS280	parity checker
12-110	9	16	4116	see note
21	1	28	I3242	address multiplexer
22-210	9	16	4116	see note
31,67	2	20	LS245	bi-directional buffer
32-310	9	16	4116	see note
41,44	2	14	LS04	hex inverter
42,47	2	14	LS00	quad 2-input NAND
43,72	2	12	LS02	quad 2-input NOR
45	1	16	LS112	dual JK bistable multivibrator
46,53	2	16	LS161	sync. binary counter
47,48	2	14	LS37	quad 2-input NAND clock driver
51	1	40	M6809A	microprocessor, 1.5MHz
52	1	16	LS139	dual 2-to-4 decoder
53	1	14	LS122	monostable multivibrator
54	1	40	WD1793	floppy-disc drive controller
55	1	40	M6821	p.i.a.
62,63	2	24	I2732	4K by eprom, T_{acc}=450ns
56	1	16	LS175	quad D bistable
66	1	16	LS157	quad 2-to-1 line multiplexer
71	1	24	M6850	a.c.i.a.
73	1	14	LS86	quad 2-input ex-OR gate
74	1	14	LS132	quad 2-input Nand, schmitt
75	1	28	EF96364	video display controller
76	1	20	LS240	octal 3-state inverter
77,78	2	18	2114	1K by 4 static ram
81	1	14	LS00	quad 2-input NAND
83	1	14	LS04	quad 2-input NOR
84	1	16	LS161	sync. binary counter
85	1	24	I2716	2K by 8 eprom, T_{acc}=450ns
86	1	20	LS273	octal D bistable
95	1	16	LS165	8-bit serial shift reg.

See note for other i.c. locations

Other components

2N2222	5	video, RS232 output transistors
1N4150	2	video, RS232 output diodes
2N2907	1	RS232 output transistor
L.e.ds	4	parity checking, high-efficiency red

6.00MHz crystal
1.008MHz crystal
DIP heeders for video end RS232 output
25-pin D-type connector for RS232 output
Single-pole two-wey switch for display-page select
Three, 16-wey insuletion-displecement connectors
Vero 07-0130A wire-wrap board
Wire-wrep pins (1 packet), wire, tool, un-wrap tool and wire stripper. **Wire-wrep sockets**:

Pins	Quentity
14	14
16	39
18	4
20	4
24	4
28	2
40	3

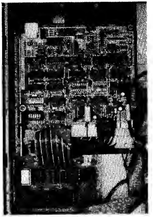
Notes
Memory circuit wes designed using Mostek MK4116-3 data sheet end most criticel timing specificetion wes T_{acc}=135ns (column-address strobe). Positions $IC_{57,82,92}$ ere 16-pin dil for plugs e,b and c respectively. Positions $IC_{91,93}$ ere also 16-pin dil for RS232 and video signals. Resistors are 10% and cepecitors are +80/−20% except where tolerances ere given.

Disc interface

Typa	Qty	Pine	Comments
LS244	1	20	octal buffer
'38	2	14	standard t.t.l. quad NAND, o.c.
LS123	1	16	dual monostable multivibretor
LS161	1	16	4-bit binary counter
LS163	1	16	4-bit binary counter
LS74	1	14	duel D bistable multivibrator
LS14	1	14	hex inverter, schmitt
LS04	1	14	hex inverter
K1160	1	14	8MHz oscilletor (Motorola)
LS138	2	16	3-to-8 line decoder

Other components
Wire-wrep socket, 14 pin (4 off)
Wire-wrep socket, 16 pin (10 off)
Wire-wrep socket, 20 pin
Wire-wrep boerd 176 by 110mm, e.g. Vero 02-0120H
34-wey insulation-displecement connector
34-way insuletion-displacement ceble to drive
Disc drive, e.g. Teac FD50A (up to 4)
Drive power connector (AMP1-480424-0)
Pins for ebove connector (AMP60617-1, 60619-1, 4 off)
Decoupling capacitors, 100n (6 off)
Decoupling capacitor, 100μ
Input resistors, 333 (4 off)
Input resistors, 220 (4 off)
Timing resistors, 30k (2 off)
Timing capacitor, 2μ 10V
Timing cepecitor, 33μ 10V

Alternetiva oscilletor components
Hex inverter, LS04
Resistor, 464 (2 off)
Cepacitor, 20p
Crystel, 8MHz



Wire-wrapped disc interface board bottom, and the disc-drive main circuit board.

Power supply

MC3405	op-amp/comparator, alternative 158 op-amp and 193 comparator
LM7812	12V, 1A regulator
2N2222	n-p-n (4 off)
2N2907	p-n-p (2 off)
2N4036	p-n-p (2 off)
2N6476	p-n-p (2 off)
2N4443	s.c.r.
1N437	ref. diode, elternative 1N960B 9V zener
1N4371	zener, 2.7V
1N4372	zener, 3V, elternative 2.7V
1N751	zener, 5.1V
1N963	zener, 12V
MR852	fast recovery diode
MDA970-2	bridge rectifier, 4A
1N4150	diode, alternative 30V switching diode, pref. Schottky
HLMP-1300	high-efficiency red led, 2.2V drop

Capacitors

1n	10%
470n	(2 off)
100n	(2 off)
22μ	10V tantalum
22μ	20V
1m	12V low equivelent series resistance, e.g. Sprague 672D046 or Dubilier UPC1052
8m	40V, alternatively 4m

Reeietors

0.13	1W
100	(2 off)
133	0.25W
200	
680	0.25W
1k	0.25W (6 off)
1.5k	
1.96k	
3.16k	(2 off)
10k	(6 off)
28.7k	
75k	
100k	(5 off)
50k	preset pot.

Transformer is a 15V r.m.s. 2A type and should be protected by e 500mA slow fuse. A mounting kit is required for the 2N6476, a cooling tab for the T05 transistor, and the toroid is en Arnold A-930157-2 with 35 turns of 21 s.w.g. (not 19 s.w.g. es on the drawing). The toroid is available from Walmore Electronics Ltd, 11 Betterton Street, Drury Lane, London WC2H 9BS.

takes too much current from the −5V supply, an indication being that the rams persistently give parity errors on power up which disappear when the RS232 terminal is disconnected. Forth response OK is preceded by the stack depth.

The problem of driving capacitive loads with l.s.t.t.l. outputs showed up as undershoot in signals passing from the interface board to the controller. Although the prototype worked with the undershoot, it was cured by taking an inverted version of the required signal back to the main board and inverting it there with a spare l.s.t.t.l. gate. Capacitance of the insulation-displacement connection between the two boards was avoided in this way. Spare connections on the inter-board connector should be grounded and ground should be placed near active signals, e.g. clocks, disc data.

Although for 8K of memory one gets a compiler and operating system and programming and execution unit there is still much to be done. I think that games are one of the best ways to learn about computers for the definition of a problem to be solved is often as difficult as solving the problem. Forth is particularly suited to games programs — the *Byte* game contest was won by a game written in Forth[10].

Reference

10. A. Saunton-Angus, Cosmic conquest, *Byte*, Dec. 1982, p.124.

Further reading

C. H. Ting, Systems Guide to Fig-Forth, Mountain View Press.
Forth Dimensions, Forth Interest Group, PO Box 1105, San Carlos, CA94070 (house magazine for members).

Brian Woodroffe has found a way of speeding up disc operations and data-transfer rates so that faster units such as the Sony Microdrive and 8in drives can be used with the Forth computer. Descriptions will follow. WW

Forth computer

Exceeding Brian Woodroffe's earlier expectations, his 6809-based Forth computer can be used with disc drives requiring high data-transfer rates – including Sony's microdrive and 8in floppy-disc drives. Access times for standard drives can also be reduced using a minor hardware modification.

by B. Woodroffe

After using the computer for a while I became discontented with the disc system because the software caused a one second delay each time access to the disc was required. This waiting time is needed to allow the drive to reach its operating speed – the software doesn't know whether the disc is running or stopped before access – and much of the benefit of the virtual memory system is lost because of this delay. Forth keeps data from a number of disc sectors in memory. When data from a sector that is not in the main memory is requested by the program, Forth overwrites one of the buffers with the required data. But if data in the buffer has been changed Forth first sends the data back to the disc so there can be two 1s delays for one disc call.

Keeping the disc constantly rotating is the easiest way of avoiding this delay but this was rejected because it shortens the life of the drive even though the motor is a brushless d.c. type. The method chosen relies on the fact that disc access operations are not uniform in time i.e., they are likely to come in bursts, especially when loading or listing screens from a disc and as a result of the virtual-memory buffer replacement algorithm described above. Keeping the drive running for a short while after a disc access is made means that it is likely that the disc will be running when the next disc access is required.

Normally, the disc-drive motor is turned off by the disc-select signal going false (p.i.a. A port, D_6, 0 to 1) when the program finishes using the drive. In the modification the drive motor enable signal is held true for five seconds after the drive-select signal goes false by a monostable multivibrator triggered by the trailing edge of the p.i.a. signal. As the software always assumes that the drive is up to speed and available, even though the monostable i.c. might have completed its cycle, a means of ensuring that the WD1793 controller doesn't try to access the drive during the motor start period is required. During this period the ready signal is held false by a further monostable multivibrator fired by the drive-motor start signal. A low-pass filter after the NOR gate combines the two sources of motor-on signal to allow for the set-up time of the 5s monostable.

This small hardware modification, consisting of two s.s.i. devices relieves the software of all considerations of motor-start latency. To prevent erroneous triggering the two monostable multivibrators should be grounded separately.

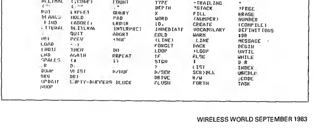
Interfacing 8in drives

I found it galling that my 1.5MHz 6809 Forth system could not be interfaced with faster 8in drives, especially as these are often available second-hand at bargain prices. I have not yet got an 8in drive but I have been fortunate enough to try one of the sub-5in drives from Sony which has the same data rate as 8in drives. There is as yet no *de jure* standard for microfloppy-disc drives[1] but within Hewlett Packard, the Sony drive is the *de facto* standard. The first problem is to build a data-service routine that services the disc at a rate better than 11μs/byte. Although nominal disc-data transfer normally takes 16μs/byte, Western Digital specify 11 and 13μs worst-case service times for write and read respectively.

The previously used software loop (*Wireless World*, June 1983) achieves far worse than 11μs, even with the M6809 direct-page register modified to make the controller i.c. accessible through direct addressing. Analysing the software loop shows that two functions are being carried out – a byte is transferred between the WD1793 controller and ram, and bytes are counted to determine when the sector operation is completed. If the second function could be dispensed with the remaining loop would be much smaller and faster. There would be a small penalty in that the ability to read from and write to consecutive sectors would be lost as no byte count is kept. The problem now is how to break out of the disc-service software loop. Fortunately the controller gives hardware help here in that the IRQ line is activated on completion of every command; the DRQ line is activated for each byte transfer. DRQ, connected to the M6809 FIRQ line, is used in the data transfer loop to make the processor clear its SYNC state, thus synchronizing controller/processor transfers operations.

Interrupt request IRQ is tied to one of the other M6809 interrupt lines so that when a read or write-sector command is complete the processor aborts its current data-transfer loop activity and commences the interrupt routine. On application of the FIRQ signal the processor does not abort the data transfer loop and carry out the FIRQ routine because the program inhibits the FIRQ interrupt by holding the FIRQ mask bit in the condition-code re-

List 1. In this design the following Forth words are available.

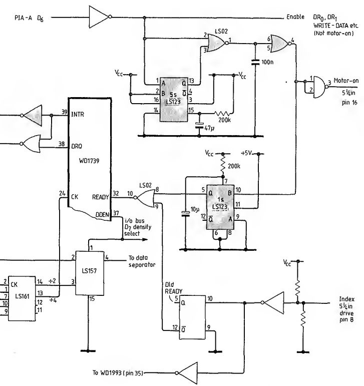


```
LIT       EXECUTE   BRANCH    0BRANCH   (LOOP)     (+LOOP)
(DO)      I         J         K         DIGIT      (FIND)
ENCLOSE   EMIT      KEY       ?TERMINAL CR         CMOVE
U<        U*        U/        AND       OR         XOR
SP@       SP!       RP!       ;S        +          D+
MINUS     DMINUS    1+        2+        1-         2-
M*        *         M/        /MOD      /          MOD
*/MOD     */        M/MOD     ABS       DABS       S->D
+-        D+-       0=        0<        LEAVE      >R
R>        R         OVER      DROP      SWAP       DUP
+!        @         C@        !         C!         TOGGLE
<BUILDS   DOES>     :         ;         CONSTANT   VARIABLE
0         1         2         3         BL         C/L
FIRST     LIMIT     USER      +ORIGIN   S0         R0
TIB       WIDTH     WARNING   FENCE     DP         VOC-LINK
BLK       IN        OUT       SCR       OFFSET     CONTEXT
CURRENT   STATE     BASE      DPL       FLD        CSP
R#        HLD       COLUMNS   HERE      ALLOT
,         C,        -         =         <
>         SPACE     MIN       MAX       -DUP
TRAVERSE  LATEST    LFA       CFA       NFA        PFA
!CSP      ?ERROR    ?COMP     ?EXEC     ?PAIRS     ?CSP
?LOADING  COMPILE   [         ]         SMUDGE     HEX
DECIMAL   (;CODE)   COUNT     TYPE      -TRAILING  "
(.")      ("        ."        DEPTH     ?STACK     ?FREE
ROT       EXPECT    QUERY     X         FILL       ERASE
BLANKS    HOLD      PAD       WORD      (NUMBER)   NUMBER
-FIND     (ABORT)   ERROR     ID.       CREATE     [COMPILE]
LITERAL   DLITERAL  INTERPRET IMMEDIATE VOCABULARY DEFINITIONS
(         QUIT      ABORT     COLD      WARM       $DR
USE       PREV      +BUF      (LINE)    .LINE      MESSAGE
LOAD      -->                 FORGET    BACK       BEGIN
ENDIF     THEN      DO        LOOP      +LOOP      UNTIL
END       AGAIN     REPEAT    IF        ELSE       WHILE
SPACES    <#        #>        SIGN      #          D.R
.R        D.        .         ?         LIST       INDEX
DUMP      VLIST     B/BUF     B/SCR     SCR>BLK    USEBLK
DR0       DR1                 DRIVE     R/W        ;CODE
UPDATE    EMPTY-BUFFERS BLOCK FLUSH     FORTH      TASK
NOOP
```

Brian Woodroffe works in research and development at Hewlett Packard.

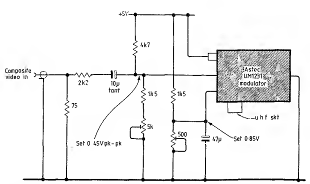
Drive	Density	WD1793 (pin 24)	Data separator
8in	Double	2MHz	8MHz
	Single	2MHz	4MHz
5¼in	Double	1MHz	4MHz
	Single	1MHz	2MHz

Additional circuits shown by tone



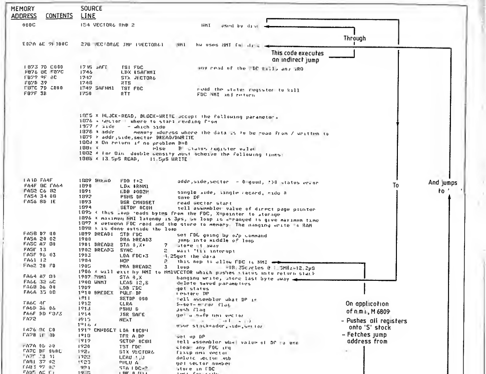
▲ *Disc interface modification speeds up overall access time by keeping the drive motor running for five seconds after the computer tells it to switch off. This significantly reduces the effect of a one second delay required for the motor to start up since disc-access operations tend to come in bursts.*

U.h.f. modulator connects to the video-controller circuit output (see June issue) so that the computer can be used with a standard tv set. ▶



gister set. The controller interrupt-request line IRQ, being connected to the processor's non-maskable interrupt input NMI, can never be masked so when this line is true the processor must be interrupted and goes to the routine requested by IRQ. The processor IRQ interrupt-request input could have been used but I wanted to leave it free for expansion.

For sector read and write operations the disc controller interrupts on transfer of the last byte. In the case of a sector-write operation the data-transfer routine is finished when the last byte is written into the controller. In sector read operations the data transfer routine is finished not when the last byte is read from the controller, but when it is written into the ram sector buffer. So when a sector is written the controller interrupts after all data transfers have taken place but when sectors are read the controller causes a jump out of the software loop before the last memory-storage operation is carried out. Worse still, latency before the controller interrupt is variable so when an interruption is made, whether or not the memory has been updated remains in doubt. The solution I chose was to code the data-transfer loop so as to maximize the time between reading data from the controller and writing it into the memory. Inclusion of a no-operation, NOP, increases the data-transfer loop time to just under the allowable maximum of 13μs to ensure that the controller always interrupts before the processor can write the byte into memory; the first operation of the interrupt routine is to write that byte into memory. Unfortunately this writing operation done outside the data-transfer loop means that the interrupt routines for sector reading and writing must be different.



```
MEMORY              SOURCE
ADDRESS  CONTENTS   LINE
 000C               154 VECTOR6 RMB 2                         NMI   used by disc

 E02A 6E 9F 000C    278 VECTOR6E JMP [VECTOR6]   NMI   hw uses NMI for disc

 F873 7D C000       1745 SAFE     TST FDC                      any read of the FDC kills any NMI
 F876 8E F87C       1746          LDX #SAFNMI
 F879 9F 0C         1747          STX VECTOR6
 F87B 39            1748          RTS
 F87C 7D C000       1749 SAFNMI   TST FDC                      read the status register to kill
 F87F 3B            1750          RTI                          FDC NMI and return

                    1875 * BLOCK-READ, BLOCK-WRITE accept the following parameters
                    1876 * sector  - where to start reading from
                    1877 * side    - which side
                    1878 * addr      memory address where the data is to be read from / written to
                    1879 * addr,side,sector BREAD/BWRITE
                    1880 * On return if no problem B=0
                    1881 *                    else    B  status register value
                    1882 * for 8in  double density must acheive the following times
                    1883 * 13.5µS READ,    11.5µS WRITE

 FA1D FA4F          1889 BREAD    FDB *+2               addr,side,sector  - 0=good, *>0 status error
 FA4F 8E FA64       1890          LDX #RNMI
 FA52 C6 82         1891          LDB #082H             single side, single record, side 0
 FA54 34 08         1892          PSHS DP               save DP
 FA56 8D 1E         1893          BSR CMNDSET           read sector start
                    1894          SETDP 0C0H            tell assembler value of direct page pointer
                    1895 * this loop reads bytes from the FDC, X=pointer to storage
                    1896 * maximum NMI latency is 3µs, so loop is arranged to give maximum time
                    1897 * between FDC read and the store to memory. The hanging write to RAM
                    1898 * is done outside the loop
 FA58 D7 00         1899 BREAD1   STB FDC               set FDC going by o/p command
 FA5A 20 02         1900          BRA BREAD3            jump into middle of loop
 FA5C A7 80         1901 BREAD2   STA 0,X+       7      store it away
 FA5E 13            1902 BREAD3   SYNC           2      wait 'til interupt
 FA5F 96 03         1903          LDA FDC+3      4.25get the data
 FA61 12            1904          NOP            2      this nop to allow FDC to NMI
 FA62 20 F8         1905          BRA BREAD2     3      loop           =18.25cycles @ 1.5MHz=12.2µS
                    1906 * will exit by NMI to NMIVECTOR which pushes status onto return stack
 FA64 A7 84         1907 RNMI     STA 0,X               hanging write, store last byte away
 FA66 32 6C         1908 WNMI     LEAS 12,S             delete saved parameters
 FA68 D6 00         1909          LDB FDC               get status
 FA6A 35 08         1910 BREDEX   PULS DP               restore DP
                    1911          SETDP 000             tell assembler what DP is
 FA6C 4F            1912          CLRA                  B=set-error flag
 FA6D 36 06         1913          PSHU D                push flag
 FA6F BD F873       1914          JSR SAFE              get a safe nmi vector
 FA72               1915          NEXT
                    1916 *                              user stack=addr,side,sector
 FA76 86 C0         1917 CMNDSET  LDA #0C0H
 FA78 1F 8B         1918          TFR A,DP              set up DP
                    1919          SETDP 0C0H            tell assembler what value of DP to use
 FA7A 0D 00         1920          TST FDC               clear any FDC irq
 FA7C BF 000C       1921          STX VECTOR6           fixup nmi vector
 FA7F 33 41         1922          LEAU 1,U              delete sector msb
 FA81 37 02         1923          PULU A                get sector number
 FA83 97 02         1924          STA FDC+2             store in FDC
 FA85 AE C1         1925          LDX 0,U++             test for side
 FA87 27 02         1926          BEQ CMNDST2
 FA89 CA 08         1927          ORB #8                set for side 1
 FA8B 37 10         1928 CMNDST2  PULU X                get the ram pointer
 FA8D 39            1929          RTS

 FFFC E02A          2525          FDB VECTOR6E   NMI
```

Notes - By changing contents of location 'vector6' the write routine exits from its loop to location 'WNMI'

- Vector6 is in ram, all other locations are eprom

Diagram of program flow during a sector read on the Forth computer.

The processor starts its interrupt sequence by pushing appropriate registers onto the stack and jumping to code pointed to by a vector in high memory. In this Forth system high memory is eprom so the interrupt vectors cannot be changed to point to different routines. I remedied this by making the interrupt vectors point to code which executes a jump to a location determined by a value stored in ram. By changing data in the ram location the non-maskable interrupt vector can be altered during program execution. This practice is unstructured and therefore unfashionable, but it is highly effective.

Normally an interrupt routine is completed by a return-from-interrupt instruction which restores processor register values to those prior to the interrupt, i.e. restore context. In this case the interrupt vector is being used as a jump instruction to jump out of the data-transfer loop so the first operation in the NMI routine is to delete saved registers (LEAS 12,S). As the controller is connected to the non-maskable interrupt line there is the potential for the occurrence of an interrupt when one is not required. To prevent this the NMI vector points to a safe routine when not in use which reads the controller status register, clearing the cause of the interrupt before carrying out a more normal return from interrupt, RTI, operation.

Extra signals to the disc drives, e.g. track 42, should be inverted and buffered using standard t.t.l. open-collector drivers (7438) as used for the WGATE signal. Extra input signals from the drive such as READY should be buffered using say two LS14 gates, and terminated as previously shown. I should have used the drive's ready signal, eliminating one of the monostable multivibrators connected to the index line but I did not. Also I connected my motor-on signal (5.25in drive, pin16) to the Sony drive head-load line, pin 14, whereas I should have used hold, HLD on pin 28 of the controller. These minor modifications were made because I still intend to use 5.25in drives as they currently offer better value for money than the Sony drives at one-off prices.

The matter of write-precompensation has not yet been resolved. I found by trial and error that for the Sony drive at least write-precompensation is not mandatory. It might be necessary for older 8in drives, and in commercial products to minimize the number of attempts to read the disc. Details of precompensation circuits are given in the Western Digital handbook and reference two.

For drives that keep the disc rotating, such as Sony's and most 8in drives, the disc speed-up hardware previously described should not be fitted but the drive should be connected with the motor-on

List 2. Forth words specific to this system.

```
Note the movement of data onto and off the stack is shown thus
      s1 s2 ..ABC.. r1 r2 r3
 Forth word 'ABC' takes two values off the stack (s2-top of stack) and produces
 three results (r3-new top of stack).
 VEMIT,VKEY,V?TER     - a user variable containing the execution vector for
                        EMIT,KEY, ?TERMINAL respctively
                        ..VEMIT..addr    ..VKEY..addr     ..V?TER..addr
 DPMAX                - a user variable containing the maximum address for the
                        dictionary
                        ..DPMAX..addr
 SMAX                 - a user variable containing the maximum depth of data
                        stack allowed for this user
                        ..SMAX..addr
 P!,P@                - a WORD to store a byte in, read a byte from one of the
                        user ports
                        data addr..P!..          addr..P@..data
 DENSITY              - a constant -0 for single density, -1 for double density
                        ..DENSITY..boolean
 DR-SEL               - a WORD to select the drive value top of stack
                        value..DR-SEL..
 DE-SEL               - a WORD to deselect the disc drives
                        ..DE-SEL..
 VERIFY               - a variable which if true causes a read after write
                        verify of disc writes
                        ..VERIFY..addr
 SIDE/DISC            - a constant returning the number of sides per disc
                        ..SIDE/DISC..value
 SEC/TRK,TRK/SIDE     - constants returning the number of sectors per track and
                        tracks per disc respectively
                        ..SEC/TRK..value        ..TRK/SIDE..value
 SIDE                 - a WORD to select the side of the disc depending on the
                        value at top of stack
                        value..SIDE..
 ?WRITE               - if disc is write-protected issue error message 10 and
                        abort execution to return to terminal mode
                        ..?WRITE..
 CMND                 - a WORD to execute "WD1793" command at top of stack
                        value..CMND..
 RATE                 - a CONSTANT which equals the disc drive stepping rate as
                        coded for the WD1793
                        ..RATE..value
 SEEK                   steps the floppy drive to seek the track that is at top
                        of stack
                        value..SEEK..
 STEP                 - a WORD to step the floppy disc, IN if TOS=1
                                                      OUT if TOS=-1
                                           same direction if TOS=0
                        value..SEEK..
 R-ADR                - reads the address mark off the disc, either returns
                        a/ flag=true if unsuccessful (=status of WD1793)
                        b/ sector_type, sector, side, track , false flag
                        ..R-ADR..type sector side track 0        or
                        ..R-ADR..true
 BREAD,BWRITE         - similar to FORTH BLOCK_READ, BLOCK_WRITE except
                        additional parameters are included, sector within a
                        track.
                        returns WD1793 status 0-succesful
                        address side sector ..BREAD..status
 SEC-R/W              - similar to FORTH R/W except no limit checking is done.
                        Returns to top of stack a flag  0=successful operation
                                                        1=crc error
                                                        2=seek error
                                                        3=not ready
                        addr sector flag ..SEC-R/W..status
 TRK-WR                 a WORD to write a whole track to disc, used in formatting
                        a disc. Takes from the stack the address of the track
                        image. Return WD1793 status, 0 successful
                        addr..TRK-WR..status
 TRK-BLD              - a WORD to build in memory a byte image of a track prior
                        to be written out by TRK-WR. Takes as input the side and
                        track number whose image is to be formed and produces
                        the address of where the image is.
                        side track ..TRK-BLD..addr
 ?DISC                - a WORD to find out what sort of disc is on the drive.
                        sets CONSTANTs B/BUF,DENSITY,SEC/TRK to suit.
                        ..?DISC..
 MS                     a WORD to delay execution by the number of milliseconds
                        that is top of stack
                        value..MS..
 SKEW                   currently a no operation word, that can be patched to
                        allow interleaving of the sectors when formatting a disc.
                        Intended usage is that it will modify the sector number
                        currently at top of stack
                        sector_no..SKEW..sector_no
```

signal permanently true, i.e. grounded.

Software issued (first revision) assumes the presence of disc speed-up hardware and includes the faster data-transfer loop. I will supply a drive pin connection list and format program for the Sony drive that can be modified for 8in drives to readers sending an s.a.e. to me at 632 Queensferry Road, Edinburgh. The Forth word BLD-TRK in eprom is only suitable for mini-floppy disc drives.

Thanks to Hewlett Packard for the use of their test equipment and Sony for the loan of a microdrive. Software used, based on the FIG model, was prepared on an HP64000 microprocessor development system.

Integrated circuits 87 and 88 were missing from last month's components list. They are 2114 static rams. In the photograph of power-supply spikes, vertical sensitivity for all but the clock signal is 0.5V/div.

References

1. J. Bovin, Floppy incompatibility, *Systems International*, May 1983, p.61.
2. J. Hoeppner and L. Wall, Encoding/decoding techniques double floppy disc capacity, *Computer Design*, Feb. 1980, pp. 127-135.

Brian Woodroffe plans to describe the Forth language in a subsequent series.

Wireless World Forth computer

Introduction, c.p.u. and memory circuits, May 1983, pp. 53-8.
Circuit description, video-controller circuit and peripherals, June 1983, pp. 55-8.
Software, disc controller and power supply circuits, July 1983, pp. 58-61.
Construction tips, August 1983, pp.44,45.

Complementary current mirror

Current mirrors with transistors of the same type of conductance are well known[1] and widely used in integrated circuits[2]. It is possible to create the configuration with similar properties using complementary transistors also, Fig. a. Accepting the usual assumptions[2] that

$$I_{C1}=I_{S1}\exp|V_{BE1}|/V_T$$
$$I_{C2}=I_{S2}\exp V_{BE2}/V_T$$
$$I_{B1}=I_{C1}/\beta_{F1}$$

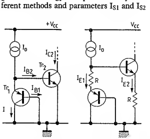
and $I_{B2}=I_{S2}/\beta_{F2}$, the output current is

$$I_{C2}=\frac{I_0}{\left(1+\frac{1}{\beta_{F1}}\right)\frac{I_{S1}}{I_{S2}}+\frac{1}{\beta_{F2}}}$$

with $|V_{BE1}|=V_{BE2}$. If the technology allows two complementary transistors with $I_{S1}=I_{S2}$, then

$$I_{C2}=\frac{I_0}{1+\frac{1}{\beta_{F1}}+\frac{1}{\beta_{F2}}}\approx I_0\left(1-\frac{1}{\beta_{F1}}-\frac{1}{\beta_{F2}}\right)$$

as in the ordinary current mirror. Usually n-p-n and p-n-p transistors in an integrated technology are produced by different methods and parameters I_{S1} and I_{S2} are not matched. But the discrete current mirror with matched resistors in emitter circuits works reasonably well, Fig. b. In this circuit



Complementary transistor current mirror with matched transistor (a) and matched emitter resistances (b).

$$I_{E1}R+|V_{BE1}|=V_{BE2}+I_{E2}R$$

and

$$I_{E1}=I_{E2}+\frac{|V_{BE1}|-V_{BE2}}{R}.$$

If transistors are designed for complementary operation, say 2N4401 and 2N4403, and emitter resistors are matched to within 1% the error is 2 to 3% without preliminary transistor matching. The gain β_{F1} of a discrete p-n-p transistor is usually high and the collector currents happen to be matched also. — I. M. Filanovsky, University of Alberta.

[1] F.J. Lidgey. Looking into current mirrors. *Wireless World*, October, 1979, vol. 68, pp. 51-58.
[2] P. Gray, R. Meyer. Analysis and design of analog integrated circuits. Wiley, 1977.

Forth language

Complementing his description of a 6809-based microcomputer, Brian Woodroffe details the language used – Forth – and why he chose it, in this second series.

by B. Woodroffe

Forth is a language well suited to modern microprocessors and is widely used in such diverse applications as word processing, data-base management, instrument and process control, video games and data acquisition. In a kernel of less than 10Kbyte the following features are provided

– An interactive system.
– A high-level compiler with all standard control features.
– Fast execution, comparable with machine code because of the compiler.
– The language system is largely processor independent; only around 20% of the code written in assembly language need be changed to suit the computer.
– Virtual memory and application-oriented program modules.

Further, the system may be readily extended to suit new applications because the compiler can be modified by the user and new data structures introduced. These features are achieved by defining a virtual machine which is easily simulated by any target machine. Using 'threaded code', transferring control in the host from one virtual machine instruction to the next is quick and easy. Instructions of the virtual machine are used to build the monitor and compiler. Using the monitor the user may examine the effect of a series of Forth instructions and using the compiler this series may be added to the instruction set for future use.

Background

Forth is a computer language for fourth generation computers[1]. The language would have been called Fourth but six letters would not fit in the IBM1130 job-control language that its inventor, C. H. Moore, was then working with. Today Moore's company Forth Inc. is foremost in marketing FORTH for many different applications, besides the field of astronomy where it first found favour[2]. Other companies such as Miller Microcomputer Services and Laboratory Microsystems sell their own versions of Forth but the prime mover of Forth in the home-computer/hobby field is the Forth Interest Group* (FIG). They have made versions of Forth available for many computers including the PDP-11 and for 8080/Z80, 6800, 8086/8088 and 6502 processors. There are many versions of Forth and while all are similar no two are necessarily identical. For example, Poly Forth, FIG Forth and Forth 79 are all Forth but they are not the same. They differ primarily because of differences in the processor on which they run (16 or 8 bit memory, port or memory mapped i/o, etc.). FIG Forth will be used in all following examples.

Forth is a collation of different sofware concepts forming a coherent whole. As an operating system, it is not as powerful as most but it takes care of all terminal and disc input and output. Small assembly-language routines must be supplied by the user to interface his hardware to the relevant system calls. It is also possible that memory-allocation changes may also have to be made. Most of Forth is written in Forth. It may seem strange that a language may be defined in terms of itself but one would use English words to explain the English language. Defining the language in this way means that programs may be transferred between different computers and implementations. There is a base instruction set which must be written in the machine code of the host computer. This is the only machine code required and the process is known as simulating a virtual Forth machine.

Most computer languages are programs which, recognizing statements in a source language, convert them into a target language. Usually the source language is text readable by humans in ASCII form and output is machine code of the computer. This is not always the case: cross compiling results in the target code being different from the host computer machine code. More exceptionally there are cases where the machine code can only be executed by a hypothetical computer, an example being O-code for the language BCPL[3] and P-Code for certain implementations of Pascal[4]. This is also the case for Forth and the virtual-machine execution mechanism will be explained first.

Threaded code

Explanation is simplified by visualizing a machine-code program for the processor concerned as a succession of subroutine calls. These calls transfer program control to each subroutine in turn. A stack, i.e., last-in-first-out list, would be the mechanism by which each subroutine returns control to the correct point in the main program. Knowing that the main program is solely a succession of calls it is now possible to reduce the main program to a list of subroutine addresses by removing the subroutine op-code, and to have a special program known as an address interpreter to transfer control down the main program address list. This is called threaded code, for the main program is the thread into and out of which the address interpreter threads control[5], List 1.

List 1. Comparisons of hard code and direct threaded code.

Normal code	Threaded code	
	Address interpreter	Thread
call A	1: ip+1 –> ip	A
call B	call [ip]	B
call C	jmp 1	C

In List 1, letters A, B and C denote machine-code subroutines, ip is the threaded-code instruction pointer and parentheses indicate one level of indirection. Threaded code trades the cost of the code for each call saved for address interpreter speed. In a long program the code cost of the address interpreter will be negligible. Further savings can be made by replacing the subroutine return statement by a jump to the address interpreter and changing the address interpreter as shown below. This releases the stack pointer used for subroutine calls and returns. It is important that the instruction pointer can be speedily accessed, for example by keeping it in a processor register, so as not to slow down the address interpreter by causing unnecessary memory activity.

If the lists are considered to be the actions of a virtual machine then a software routine NEXT represents the hardware execution fetch of the virtual machine. In a threaded-code computer the time of interpreting these lists is dominated by the time of the NEXT operation so it is best to run threaded code on a computer that handles NEXT efficiently or to use microcode.

Code routine including return

```
A: xxx
   jmp NEXT
```

New address interpreter

```
NEXT: ip+1 –> ip
        jmp [ip]
```

Indirect threaded code

The next improvement is to allow called routines to be not just pure machine code but also address lists. This is done by having a special routine that knows that the following data in the list are not code but addresses that must again be interpreted. Further, the routine must suspend interpretation of the main program while interpreting this new list of addresses. Return of control to the suspended list is done using a stack to save and restore the instruction pointer which is similar to the machine-code subroutine call/return operation. There must be an equivalent code routine to return control to the main list.

Normal code routine

```
A: machine code
    . . .
    jmp NEXT
```

*Forth Interest Group, PO Box 1105, San Carlos, CA94070, USA.

```
Threaded routine
P: sp−1 −> sp
   ip     −>[sp]  (push current ip)
   #L−1 −> ip   (start interpreting new
                 list)
   jmp NEXT
L: A             (code routine)
   B
   C

Return routine
[sp] −> ip       (pop ip)
sp+1 −> sp
jmp NEXT
```

As most routines are likely to be lists and not machine code this stacking method, similar to subroutine calling, will take a lot of code area. Considerable space would be saved if there was just one copy of this routine. The address interpreter would normally jump to this routine but it would also have to execute code routines. This is done by making the first element of each list a pointer to code rather than the code itself. In the case of lists the pointer points to the stacking operator but with code routines it points to the next code address.

```
New address interpreter
NEXT: ip+1 −> ip
        [ip] −> w
        jmp [w]

Stacking operation
DOCOL: sp−1 −> sp
        ip     −> [sp]
        w+1 −> ip
        jmp NEXT

Destacking operation
SEMIS: [sp]  −>ip
        sp+1 −> sp
        jmpNEXT

Code routine
A: $+1   (point to next location)
   xxx
   . . .
   jmp NEXT

List routine
DOCOL
P
Q
SEMIS
```

This is the equivalent of machine-code subroutine call and return instructions. In Forth, the stacking and destacking operations are called DOCOL and SEMIS respectively. At the beginning of each address list, the extra address introduces a level of indirection – this is indirect threaded code[6]. In Forth the lists are divided into two parts, one being the code field which points to the address and the other known as the parameter field where the code is. These two parts and dictionary data, to be described, form a WORD. Code pointed to by the code field determines how the parameter field is interpreted. In the case of code words, the code field points to the parameter field. When the code field points to DOCOL, the parameter field is to be interpreted in a similar way to a subroutine. It is possible for the code field to point to some other routine which may make different use of the parameter field. Two examples of this in Forth are DOCON and DOVAR. The former treats the value in the parameter field as a constant and pusbes it onto the data stack, to be described, whereas DOVAR pushes the address of the parameter field which is used as the storage location for that variable. To enable these routines to access the parameter field a third register, known as 'w', is required.

The address interpreter for indirect threaded code is more complicated than that for direct threaded code and so it is even more important to choose a processor with a suitable instruction set. Surprisingly for direct threaded code, NEXT can normally be coded using the processor subroutine-return op-code provided that the processor uses a stack that may be placed anywhere in memory. As the stack pointer is pointing to the thread, the processor must not receive interrupts for the status cannot be saved without destroying the thread. NEXT for indirect code is more complicated as it involves an extra level of indirection.

Choosing a processor, stacks and language-control structures are subjects of the next Forth language article.

An i.c. in the Forth computer switch-mode power supply on page 61 of the July issue was incorrectly designated the MC3045. The correct designation is MC3405.

References

1. C. Moore, *Forth dimensions*, vol. 1, no. 6, FIG

2. C. Moore, *Astronomical Astrophysics Supplement*, 1974, vol. 15, pp.497-511

3. M. Richards, The portability of the BCPL compiler, *Software Experience and Practice*, 1976, vol. 1, pp.135-146

4. D. Barron, Pascal, the language and its implications, Wiley, 1981

5. J. Bell, *Communications of the ACM*, vol. 16, no. 6, pp.370-372

6. B. Dewar, *Communications of the ACM*, vol. 18, no. 6, pp.330-331

WW

Glossary

Machine code. The representation, usually in hexadecimal, of the instruction and data encoding that is understood by the computer.

Assembly code. A human readable form of machine code. There is a one-to-one correspondence between assembly code and machine code.

Instruction fetch. A computer works by successively fetching and executing instructions. The instruction fetch is made from the location pointed to by the program counter. The program counter is incremented one instruction at a time unless a jump (branch etc) occurs.

Virtual machine. At any level of analysis the computer will have a repertoire of instructions that it can execute. This is normally the machine/assembly level instructions. However by running a program on this machine it can be made to look as though it has a different instruction set. It is possible to time share the computer between two or more users so that they both think they have a separate computer. These techniques are known as creating a virtual machine.

Op-code. Each different instruction is encoded into a unique symbol (usually binary) known as an op-code.

Host computer. The computer on which the program is currently executing.

Target computer. The computer on which the program being developed will execute.

Cross compilation. A cross compiler runs on one machine and produces output for another. Host and target machines have either different op-code encodings or instruction sets, or both.

Compiler. A program that recognizes that the input language agrees with a defined grammar. If it agrees it will usually produce an output in some other defined language, or error messages as to why the input is not in the source language. Normally input is English-like, (e.g. Forth/Pascal) and output is machine code.

Microcode. Microcode is a mechanism used to build computers to understand machine-code instructions. Within a microcoded computer there is another computer with its own microcode instruction set. By writing new microcode, the assembly-level machine can be made to have new instructions.

Kernel. A central program on whose resources all application program rely and interface to.

Operating system. A computer program which manages the computer's resources. It will take care of all input-output etc. so that the application programmer need not worry about how to get characters to and from a terminal, etc.

Software driver. A small program specific to each input/output device that is included in the operating system.

Terminal. Visual-display unit and keyboard, teletype.

Indirection. An addressing mechanism. An instruction requires data to act upon – the instruction gives details of how to find that data. Normally it will give the address of the data, but in the cases of indirect addressing it will give the address at which the address of the data may be found. That is one level of indirection. Up to three levels, ie the address which contains the address which contains the address which contains the data, are common.

Call. A subroutine call is a mechanism whereby machine-code execution is temporarily suspended while the subroutine is executed. Execution will restart at the instruction after the call when the subroutine finishes. The restart address (return address) is often kept on a stack.

Code field. A part of a Forth Word definition. The contents of the code field always point to machine code of the target machine.

Machine. Computer (state machine).

Monitor. A program that monitors user requests (as typed in at the terminal). Usually gives message (<OK>) when the command has successfully executed. Monitor is also the name given to a technique used in real-time programming developed by C.A.R. Hoare et al.

Virtual Forth machine. The assembly-language programmer creates a virtual machine that executes low-level Forth instructions.

Virtual-machine execution mechanism. The means by which the assembly-language programmer makes the virtual Forth machine transfer control from one Forth instruction to the next.

Forth language

Selecting a processor to suit the language, and control structures are subjects of Brian Woodroffe's second article illustrating why he designed his computer around Forth.

by B. Woodroffe

Forth's speed is directly related to how efficiently the computer can execute the NEXT operation. The Table shows how NEXT is coded for some popular eight-bit microprocessors, the 6809 processor executes the operation quickly so a NEXT operation may be included at the end of code routine. This improves performance since the 'JMP NEXT' operation needed for most processors is avoided – in stark contrast to conclusions drawn from one manufacturer's benchmark tests.

NEXT is the virtual-machine instruction fetch so the choice of a processor to run Forth on should be dominated by speed and memory costs of the NEXT operation. Further, 6809 registers exactly match those required for Forth as can be seen in List 2. Machine code in the host computer represents the Forth machine, the Y register taking on the role of the Forth program counter. Following examples of simulating the virtual machine, in 6809 machine code, confirm that this processor is well suited to Forth.

The stack

So far, only the control mechanism by which Forth transfers control from one word to the next has been described, but the language must also control and manipulate data. This, too, is done by means of a stack, but this storage area is known as a data stack, as opposed to the one previously described which is known as the 'return' or 'control' stack. Separation of the stacks simplifies things, normally, data and control operations use the same stack. The stack is further broken down into 'frames' with markers to denote which part is what. In Forth all operators, such as the words + and AND, may remove instructions from the stack, destroy them, manipulate them and push results back onto the stack many times. This has the advantage that operators need not be told where their operands are, which results in less code. A computer operating this form of addressing is known as a zero-address machine, for operand addresses are implicit in the instruction. These words may be in the machine code of the target computer or determined using words already defined.

List 2. Registers of the 6809 suit Forth requirements

6809 register		Forth usage	
S	stack pointer	RP	return stack pointer
U	user stack pointer	SP	data stack pointer
Y	index register	IP	instruction pointer
X	index register	W	current c.f.a
D	accumulator	–	accumulator

Using a stack avoids problems caused by parentheses and operator precedence. As far as the computer is concerned the problem is solved, List 3, but programmers used to infix notation may find postfix notation (reverse-Polish notation) difficult, e.g.

Postfix	Infix
34+56+×	(3+4)×(5+6)

Table. Coding and performance analysis of the Forth NEXT operation for popular eight-bit microprocessors.

Processor	6809	6800	Z80/8085	8088	6502
Code	LDX 0.Y++ JMP [0.X]	JMP NEXT LDX IP INX INX STX IP LDX 0.X STX W LDX 0.X JMP 0.X	JMP NEXT LDAX B INX B MOV L.A LDAX B INX B MOV H.A MOV E.M INX H MOV D.M XCHG PCHL	JMP NEXT LODS AX MOV BX.AX MOV DX.BX INC DX JMP WORD PTR [BX]	JMP NEXT LDY #1 LDY [IP].Y STA W+1 DEY LDY [IP].Y STA W CLC LDA IP ADC #2 STA IP BCC L INC IP+1 L JMP W−1
Memory bytes	4	17	14	11	28
Processor clock cycles	14	44	60	58	43
Normal cycle time (µs)	1	1	0.25	0.2	1
Total time (µs)	14	44	15	11.6	43
Memory-access (ns)	695	530	250(Z80)	450	650
Time for 450ns-access memory (µs)	9	37	27	11.6	29.7
Speed relative to 6800*	4.11	1	1.37	3.19	1.25

*Value rises proportional to speed.

List 3. Some 6809 code arithmetic routines including add, subtract and two's complement

```
'+'     FDB $+2
        PULU D
        ADDD 0.U
        STD 0.U
        NEXT
MINUS   FDB $+2
        LDD #0
        SUBD 0.U
        NEXT
@       FDB $+2 (fetch)
        LDD [0.U]
        STD 0.U
        NEXT
!       FDB $+2 (store)
        PULU X
        PULU D
        STD 0,X
        NEXT
DUP     FDB $+2
        LDD 0.U
        PSHU D
        NEXT
OVER    FDB $+2
        LDD 2.U
        PSHU D
        NEXT
SWAP    FDB $+2
        PULU D.X
        EXG D.X
        PSHU D,X
        NEXT
DROP    FDB $+2
        LEAU 2,U
        NEXT
```

NEXT is defined as a macro instruction.

Parameters are also passed between separate lists using the stack. The word consumes as many stack elements as required and pushes back its results. Some defined Forth words for subtracting and doubling the top of the stack respectively are

```
'−'FDB DOCOL     "2*"FDB DOCOL
   FDB MINUS         FDB DUP
   FDB ADD           FDB PLUS
   FDB SEMIS         FDB SEMIS
```

Language control structures

As has been shown, Forth passes control from one item in a word to the next and results are calculated. These words can be either machine-code words or pointers to other words. How control may be diverted to form if-then-else or repeat-until structures is the following subject, starting with an explanation of how Forth tests for true or false conditions by simply considering a non-zero value at the top of the data-stack as a true condition. Examples of conditions that create these flags are '0=', '0<', '=' and '<' in the form of code words or Forth words, as appropriate, Lists 4, 5. Diversion of control is carried out by Forth

List 4 Code routines leaving a flag at stack top

```
0EQUAL  FDB $+2
        LDD #1      assume true (i.e.
                    zero)
        LDX 0,U++   get operand, set
                    6809 flags
        BEQ 0E1
        DECB        was <>0 so set
                    Forth flag
0E1     STD 0,U     put back Forth flag
        NEXT

0LESS   FDB $+2
        LDB #1      prepare true
        LDA 0,U     get sign to A
        BMI 0L1     -?
        CLRB        no, leave false
0L1     CLRA
        STD 0,U
        NEXT
```

List 5. Forth routines leaving a flag.

```
"="     FDB DOCOL
        FDB SUB
        FDB 0EQUAL
        FDB SEMIS

"<"     FDB DOCOL
        FDB SUB
        FDB 0LESS
        FDB SEMIS

">"     FDB DOCOL
        FDB SWAP
        FDB LESS
        FDB SEMIS
```

words BRANCH and 0BRANCH, the former taking the next storage cell as a branch offset and the latter branching or not depending on the value at the top of the stack. If the flag is false, the threaded-code instruction pointer, ip, is incremented by the offset value contained in the next program storage cell. When the flag is true, this offset is skipped and execution continues with the next word. Controlled loops may also be constructed. Using 'begin ... until' structures, statements between are executed so long as the flag at the top of the stack remains false. Iterative loop type structures such as '100 TIMES DO' are handled by taking initial and limit loop indexes off the data stack and storing them on the control stack. At the potential end of the loop the current index is incremented and compared with the limit. If the limit is exceeded a branch is executed as described above, otherwise the indexes are deleted and the offset skipped to continue execution, List 6.

List 6. Code for diverting control flow if the flag at the top of the stack is false.

```
0BRANCH   FDB S+2     6809 code
          LDD ,U++    test and delete
                      Forth flag
          BNE 0B1     <>0, branch if true
          LDX 0,Y     get jump offset in
                      X
          LEAY Y,X    add offset
          NEXT

0B1       LEAY 2,Y    skip over offset
          NEXT

BRANCH    FDB S+2
          LDX 0,Y
          LEAY Y,X
          NEXT
```

```
ROOTS                     ( stack : c b a start defining new word 'ROOTS' )
  SWAP MINUS              (       c a -b )
  OVER                    (       c a -b a )
  DUP +                   (       c a -b 2a quicker than 2* )
  /                       (       c a -b/2a )
  ROT ROT                 (       -b/2a c a save -b/2a )
  /                       (       -b/2a c/a )
  OVER DUP*               (       -b/2a c/a -b/2a*-b/2a )
  +                       (       -b/2a b**2/4a-a/c )
  DUP 0<                  ( is top less than 0, ie imaginary roots? )
  IF                      ( test flag )
    DROP DROP             ( delete partial results, send <cr><lf> to terminal )
    CR ." imaginary roots" ( and print message )
  ELSE
    CR                    ( real roots, send <crc><lf> to terminal )
    0                     ( convert 16-bit positive number to 32 bits )
    SQRT                  ( get back square root )
    OVER OVER +           ( duplicate both parts of answer and get 1st result )
    ." roots are" .       ( print message and first answer )
    ." , and" - .         ( print message and other answer )
  ENDIF                   ( continue execution )
  CR ;                    ( send <cr><lf> and stop compiling return to execution )
```

List 7. Forth code used to calculate the roots of a quadratic equation. The stack is represented across the page with the top of the stack at the right.

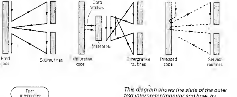
Using Forth

List 7 is an example of a Forth routine for calculating the roots of a quadratic equation, given that the indexes are on the stack. Forth has the shortcoming that it only handles integer arithmetic so non-integer results will be incorrect. The program example illustrates a number of Forth concepts, e.g., stack manipulation, passing parameters and terminal output. Words used in the program are explained in the next article, as are the dictionary and compiler.

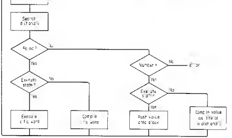
Reference

7. Intel iAPX88 Book, July 1981, appendix pp. 20-36

Three flow diagrams compare, from left to right, hard code, interpretive code and threaded code.



This diagram shows the state of the outer text interpreter/monitor and how, by manipulating states, Forth words can compile, yet still have access to the execution power of previously defined words.



Forth language

Forth consists of words and new words must be compiled and entered into its dictionary. Following a description of the dictionary and compiler, Brian Woodroffe discusses advanced concepts in this third article.

by B. Woodroffe

Having shown how the address interpreter executes lists of addresses to execute program commands and that threaded code is compact, I will now explain how Forth builds these lists, i.e., how it compiles. Each list representing an action is rather like a verb in a natural language and in Forth is called a WORD. The collection of these words, which is Forth, is known as the dictionary. The outermost WORD in Forth breaks input text down into character strings which it then searches for in the dictionary. (Spaces are important, for instance, '−1' is treated as a negative number, whereas '− 1' is treated as the arithmetic subtraction operator followed by the positive number one.) If the string is found, it is executed, otherwise an error message is generated. The dictionary also needs a mechanism to allow the search to occur. Searching involves a traverse of all the words in the dictionary. Each entry has a pointer to the previous one (link field), which makes the dictionary a linked list. To enable matching of the source text each word also has its name in ASCII form (name field). Dictionary entries for each word, List 1, have four fields – name field, link field, code field and parameter field. The name field also contains data for use by the compiler (precedence and smudge bits) as will be explained, and it includes the length of the name to allow variable name lengths of up to 31 characters.

For language expansion it is important to be able to build dictionary entries for new words. This is done by invoking the compiler. When the compiler is invoked (Forth word ':'), the language state is switched from execution to compilation. Next, input text is scanned forward for the next text string which is used to build a newly created name field. The name is 'smudged' so that during the building of the incomplete definition, the same name cannot be found. This normally prevents recursion, but again in Forth, this rule can be overcome, List 2. Then the linked list of the dictionary is updated by copying it into the dictionary link and the address of DOCOL is copied as this new word's code field. Next, input text is scanned for character strings. As these character strings are matched with words that already exist in the dictionary, the code field of each word found is copied into the parameter field of the word being compiled. Finally, as the word to terminate compilation is encountered ';' SEMIS is copied as the last word of the definition and the Forth program is returned from the compile state to the execution state. The compiled word is now 'unsmudged' to allow it to be accessed.

The compile process can be quite long as many dictionary searches have to be made. As the dictionary is a linear list and the code routines which ultimately have to be compiled are at the bottom, it is a long search. No speed up algorithms such as hashing have been applied to standard FIG Forth though there has been experimentation[8,9,10]. As so much work is done during the compile phase the execution performance of newly defined words is nearly as quick as predefined words. Further, as the first half of the dictionary entry

List 1. Each dictionary entry has four fields called name field, link field, code field and parameter field.

```
        Dictionary entry
        7 6 5 4 3 2 1 0
NFA     1 p s <1 e n >     p=precedence, s=smudge, len=length of name
        0 a s c i i 1
        0 a s c i i 2      ASCII characters of word
        0 . . . . . .
        1 a s c i i n      d7 set on last character of name
LFA     <             >    16 bit address of previous n.f.a.
CFA     <             >    16 bit address of code routine
PFA     <             >
        <             >    parameters, normally other c.f.as
        <             >
```

List 2. Example of recursion in Forth to calculate a factorial.

```
First define
: MYSELF
    LATEST                ( put address of word currently being defined on stack )
    PFA CFA ,             ( convert to code-field address and compile )
                          ( it in dictionary so that it may call itself )
  ; IMMEDIATE             ( as this word is to execute when in the compile state it has
                          'precedence'. )
Then use myself in the recursive definition
  : FACTORIAL ( n ... )
    DUP 1 = IF ELSE       ( end of recursion?, yes leave 1 as 1!=1 )
    DUP 1 -               ( n, n-1 ... )
    MYSELF                ( call myself to calculate n-1! )
    *                     ( n!;=n*[n-1!] )
    ENDIF
    ;
```

List 3. Definitions of IF, ENDIF and ELSE.

```
: IF
  COMPILE 0BRANCH         ( compile into the dictionary the c.f.a of '0BRANCH' )
  HERE                    ( place on stack where we are )
  0,                      ( make space for jump offset )
; IMMEDIATE               ( make compiler execute this word, even if in compile state )

: ENDIF
  HERE                    ( where are we? )
  OVER -                  ( calculate offset to HERE executed in IF )
  SWAP !                  ( patch in offset to address left by IF )
; IMMEDIATE

: ELSE
  COMPILE BRANCH          ( compile run time address of routine to skip false statements )
  HERE 0 ,                ( make space for jump offset )
  SWAP                    ( get address of where IF was )
  [COMPILE] ENDIF         ( use ENDIF to fix address, ENDIF is immediate and to overwrite
                          that such that it is compiled )
; IMMEDIATE
```

List 4. Examples of VARIABLE and CONSTANT.

```
: VARIABLE
    <BUILDS          ( variable is a new parent word )
    ,                ( store in the p.f.a. the value that was top of stack )
    DOES>            ( start defining what offspring will do )
    ;                ( nothing – p.f a. is storage location for an offspring of type VARIABLE )
: CONSTANT
    <BUILDS ,
    DOES> @          ( constants provide a constant value which has been )
    ;                ( stored in their p.f.a. )

0 VARIABLE ABC       ( ABC is an offspring with initial value 0 )
1000 CONSTANT K      ( K is an offspring of value 1000 )
```

Forth words used in last month's program example

Stack operators

DUP pushes the top of the stack on to itself so the stack increases by one and the top and second stack elements are equal.

SWAP interchanges the two elements at the top of the stack.

OVER places a copy of the second element in the stack on the top. The original top element is now second and all other elements move down one.

ROT takes the third element and makes it the top element so the old top element becomes second and the old second element becomes third.

Arithmetic operators

MINUS replaces the top stack element with its two's complement.

+ takes off the top two elements, adds them and pushes back the result which becomes the new top of stack.

– takes off the top two elements and pushes back the result of second element minus the top element.

* takes off the top two elements, multiplies them and pushes back the result.

/ takes the top two elements, divides the second by the first, and pushes back the result.

n pushes the value of n on the stack.

SQRT takes the top two elements, treats them as double precision (32-bit), and pushes back the 16-bit square root.

0< if the top stack element is less than zero it is replaced by a one (true), if it is not less than zero it is replaced by a zero (false).

Control operators

IF tests and deletes the top stack element and executes the next word if the condition is true. If the condition is false, control skips to ELSE or **ENDIF**, whichever comes first.

ELSE marks the end of the IF-TRUE clause and the beginning of the IF-FALSE clause.

ENDIF is the end of an IF clause.

Terminal operators

CR sends a carriage return and line feed signal to terminal.

." starts text entry into the dictionary (until "). When executed it will type out the text string.

. converts the top of the stack to an ASCII string and types it out.

(name and link fields) is only required during compilation for fixed applications where compilation is not required, these fields can be deleted. This dramatically reduces the memory requirements of the Forth system[11], and can be especially useful when the code will be placed in rom.

To enable the compiler action of Forth to do more than just that described above certain words need to execute even when the language is in the compile state. This gives the compiler the full capabilities of Forth. These words are generally involved in building control structures for the compiler (IF-ELSE-ENDIF, see List 3). These words have a precedence bit set which the compiler recognises when it matches the input text so instead of compiling its code field it executes it. In the case of IF the compiler compiles into the dictionary the c.f.a. (code-field address) of **0BRANCH** and advances the dictionary pointer to allow the as yet unknown offset to be placed. It also pushes this address onto the stack so that when the compiler encounters an ELSE or ENDIF statement it can calculate the offset back to the IF statement and store the offset there. This shows the power of Forth in that the computational ability of the language is available to the compiler and to the user. Further there are times when words that would normally execute (i.e. have precedence) need to be compiled (i.e. execution action delayed until the word currently being defined executes). This is done using the word [COMPILE]. Again, it is sometimes required to delay compilation of a word until the word that contains it executes. This is done using the word COMPILE.

Advanced concepts

With an idea of how the inner interpreter (address interpreter) and the compiler (text interpreter) work we can now move on to advanced concepts including extension, vocabularies and virtual memory. Forth is either in the compile state, when it finds words and copies their code-field addresses into the dictionary to form new entries, or in the execute state, when it executes each code-field address encountered. I have shown how certain immediate words can override the state, and can even execute in the compile state. It is also possible to overrule words which are declared as immediate and compile them, as in the case of ELSE which was described earlier.

In Forth, even the compiler can be modified. Not only can new compiler control structures be introduced but also new compiler words may be defined. Normally the programmer would have to rewrite the compiler but with this feature, known as extensibility, modification is relatively simple. It involves use of the words <BUILDS and DOES> to define a new class of words. The defining word defines words of this new class. Behaviour of the new defining word is determined by the words between <BUILDS and DOES>, i.e. when a word of the new class is defined, behaviour of the new word during compilation is determined by what comes between <BUILDS and DOES>. When a word of this new class executes, it executes the words following DOES>. To allow the parent class-defining word to access its offspring'(class-defining word to access its offspring (class-defined word), the parameter-field address (p.f.a.) of the latter is placed on the data stack. Two simple examples from the Forth compiler are VARIABLE and CONSTANT, List 4. These can easily be expanded to form arrays and tables. The word ':' is also a defining type. When offspring of ':' are executed they call the word DOCOL which decides how to execute their parameter field. An alternative to DOES> is used to define ':'; the assembler is invoked so that the parent-word execution field is machine code and not Forth but in other respects it is the same.

The major part of Forth is the dictionary, and to enable different problems to be solved in different areas of the dictionary each problem is given its own vocabulary. The dictionary may have many vocabularies alongside the normal basic set of FORTH, ASSEMBLER and EDITOR. Using vocabularies means that the same word may have different meanings, depending on which vocabulary is active. FIG-Forth has two active vocabularies – CURRENT and CONTEXT. The former is the one in which words are defined and the latter is the one which is searched first. All vocabularies are linked to FORTH (Forth's definition in Forth). Much debate is taking place on the subject of vocabularies concerning the subject of searching vocabularies[11–13].

Virtual memory

Memory is the most precious resource of a computer and although Forth makes very efficient use of it, there are still times when programmers wish it was infinite. Disc memory is much cheaper than semiconductor memory but it is also slower. By the concept of virtual memory, the memory space available to the programmer is expanded beyond the main memory to include disc storage so memory capacity as far as the programmer is concerned is only limited by the capacity of the disc. In Forth, the virtual-memory cancept is only applied for data whereas in most processor applications (e.g INS16000 series) it is also applied for program storage. Through use of the word BLOCK, the programmer can visualise the disc memory as processor memory. The Forth operating system recovers data from disc and places it in a buffer to make it accessible to the user program. Many such buffers exist in the host processor and BLOCK uses an algorithm that determines whoch blocks should be maintained in the store and which should be written back to the disc. With the right algorithm the number of disc accesses will be minimal and the apparent memory-access time low.

List 5. Representation of algorithm used for benchmark test, see Table 1.

```
8190 CONSTANT SIZE
0 VARIABLE FLAGS SIZE ALLOT              ( allocate 8191 bytes for an array )
: DO—PRIME
  FLAGS SIZE 1 FILL                      ( fill array with '1', <true> )
  0                                      ( counter )
  SIZE 0 DO                              ( set up a DO loop of 8190 times )
  FLAGS I + C@                           ( I is loop counter, get relevant flag )
  IF                                     ( C@, C! are byte versions of @,! )
              I DUP + 3 + DUP I +        ( stack is . . . count, prime, K )
    BEGIN                                ( begin a block )
      DUP SIZE <                         ( array index < size ? )
    WHILE                                ( test flag, to see if exit block )
    0 OVER FLAGS + C!                    ( set relevant flag false, FLAGS[K] )
    OVER +                               ( K:=k+prime )
    REPEAT                               ( end of block, loop back )
  DROP DROP 1+                           ( delete prime, K, one extra prime found )
  THEN                                   ( end of IF )
LOOP                                     ( end of DO . . . LOOP block )
 ." primes" ;                            ( print number of primes )
```

Space and time

I have shown how the Forth dictionary is created (i.e. its form), how it may be extended by compiling and how any processor may readily simulate the virtual Forth machine by means of indirect threaded code. As mentioned earlier, by introducing the concept of threaded code, execution speed is traded for code space. So one can expect that Forth is not as fast as the host processor's own code although it may approach it where many subroutine calls are made. Execution of a process defined in the source language divides into two parts; one is the examination of source text to find out what action is to be taken and the second is execution of the actions by the processor. In most systems the first part is carried out by compiling source text in the machine code of the host machine. In Forth this means compiled into the thread, the machine code of the hypothetical machine. Target machine code is then run in Forth using the address interpreter. Running time can be traded for space by choosing an intermediate language of suitable complexity. Running time performance of compilation has no effect on the running time of the application program, which leads to the view that the compiler should do as much work as possible. Unfortunately, compiling to machine code using a simple processor with limited addressing capability, such as a current 8-bit microprocessor, often results in the code not fitting into the memory so an intermediate target code is chosen, with the accompanying penalty of interpreting it. Forth's address interpreter costs some tens of percent.

Other losses occur because microprocessors are not zero-address devices so the zero-address function has to be simulated. Memory-space benefits are illustrated by the amount of memory required for a Forth system, which is typically 8Kbyte (may be rom) for virtual-machine simulation, the Forth compiler, i/o drivers, etc., and 8K for stacks, virtual-memory buffers and the user dictionary.

Forth is also fast because of the explicit use of the stack. In languages using the assignment operator, data normally resides outside the stack. It is brought to the stack, operated on, and finally placed back into the store. If the next statement uses the same variable it is once again taken from the store and placed on the stack. When computing partial results this causes excess memory traffic. Unless optimization is used this redundant memory activity will cause delays. Forth avoids this because normally data only resides on the stack. No unnecessary memory space or time is taken up by temporary variables.

It is interesting to compare Forth's performance with the commonly used language for microprocessors, Basic. Systems using Basic have little compiler action, the source text being saved in memory, although the key words are converted into internal tokens. During program execution, each token is parsed and acted upon in turn so the source of Basic's execution-time interpreter is close to that of the source text whereas Forth's source for the running-time interpreter is close to the language of the host computer. As all the work in a Basic system is done while the program is running the speed penalty is high, usually at hundreds of percent. Further, since Forth compresses object code into 16-bit addresses (code-field addresses are the equivalent of tokens) it is as efficient as Basic in terms of memory space.

Processing speed is an emotive issue without benchmark tests and unbiased benchmarks are notoriously difficult to produce. Table 1 was derived using the Seive of Eratosthenes (see List 5) and seems fair[14]. Qualitatively, it confirms what one could expect — assembly-language is faster, followed by compiled languages with interpreted Basic well behind. The table also shows how well the 6809 compares with newer and more popular designs and that it compares with at least one 16-bit device, the 8086. I would attribute this to the instruction set as was shown in the analysis of Forth word NEXT. A more elaborate, special-purpose instruction set does not necessarily lead to a more effective processor. This has been shown in recent research into reduced instruction set computers.

continued on page 61

Tabla 1. Relative spaeds of various processor and languages.

Processor/language	Time in seconds
CRAY-1 Fortran	.11
68000 assembly language (8MHz)	1.12
PDP11/70 C	1.52
VAX 11/780 (C/Fortran/Pascal)	1.5-5.0
8088 assembly language (15MHz)	4.0
6809 assembly language	5.1
PDP11/40 C language	6 1
Z80 assembly language (4MHz)	6 8
6809 IMS Pascal compiled (2MHz)	8.9
PDP11/70 Decus Forth	11.8
Z80 Microsoft Basic compiler	18.6
8088 Pascal (Softech compiler) (15MHz)	19.4
68000 Forth (8MHz)	27
6809 FIG Forth (2MHz)	45
8088 FIG Forth (15MHz)	55
8086 FIG Forth (12.5MHz)	64
6809 FIG Forth (1.5MHz)*	67
Z80 Forth (Timin) (4MHz)	75
Z80 Forth (Laboratory Microsystems) (4MHz)	78
Z80 FIG Forth (4MHz)	85
6809 IMS Pascal P-code (2MHz)	105
6809 Basic 09 (2MHz)	238
6502 FIG Forth (1MHz)	287
6809 TSC Basic (2MHz)	830
Z80 Microsoft Basic	1920
APPLE integer Basic	2320
TRS80 Microsoft Basic	2250
6809 Computerware Basic	4303

* Used in my design as described in *Wireless World*

continued from page 55

List 6. Array boundary checking using <BUILD . . . DOES>.

```
: ARRAY                                         ( low high . . assumes low <high )
 <BUILDS
 OVER – SWAP OVER                               ( delta low delta )
 SWAP , ,                                       ( store low is p.f.a., delta as p.f.a +1 )
 DUP + ALLOT                                    ( that much storage, byte address machine )
 DOES>
 DUP ROT                                        ( . . p.f.a. p.f.a. index )
 SWAP @ – DUP 0<                                ( . . p.f.a. required-delta flag )
 IF ." array bound error, too low" QUIT THEN
 OVER 2+ @                                      ( . . p.f.a. reqd allowed )
 OVER <                                         ( . . p.f.a. reqd flag )
 IF ." array bound error, too high" QUIT THEN
 DUP +                                          ( word index to byte index )
 + 4 +                                          ( add index-skip parameters, leaving array
                                                  address )
;
```

Forth problems

So far, only advantages of Forth have been discussed but it has some disadvantages. The most obvious of these is notation. For the beginner, reverse-Polish notation and the lack of an assignment operator (:=) are considerable problems. Practice lessens the problems though program comments and stack diagrams generally remain necessary to show what is going on.

Floating-point arithmetic is not standard and all data manipulation assumes 16-bit two's-complement arithmetic, but it may be programmed in[15]. This shows Forth's origin in the control field of computing. However, many Forth programmers maintain that most problems can be reduced to scaled-integer arithmetic. This drawback makes one aware of the processing cost of floating-point arithmetic. Forth does not use 'data typing'. This means that integer operations are used when logical operations are being performed ('0=' for NOT). There are also separate operators for 32-bit arithmetic. Computer languages can usually apply different operations for the same operator by data typing.

A more serious drawback is the lack of built-in data structures – not that Forth is any worse in this respect than Basic or Fortran. What is lacking are the type of data structures available in Pascal. In common with the formerly mentioned languages, Forth lacks a method of checking for overflow and array boundary conditions in normal operation. But as shown in List 6. This can be programmed in. Naturally, this process increased execution time but when the application works a simpler version of array can be coded by missing out the check. Finally there is as yet no file management software. Access to disc information has to be done using BLOCK numbers.

Summary

I have shown that the programmer is released from the instruction set of the host computer with little time penalty by applying threaded code. Using the compiler, one may easily extend the Forth instruction set to suit one's own application. As the whole dictionary is available all of the time (ranging from virtual-machine instructions to <BUILDS . . . DOES> structures) the programmer can tackle low or high-level problems, such as i/o driving or word processing, with equal ease and efficiency. The consistent nature of the compiler and text interpreter allow easy interactive testing of code before it is compiled. Reverse-Polish notation simplifies the compilation process and allows it to be completed in one pass in a small memory. Virtual memory and vocabularies further enhance Forth by offering infinite data space and better control of the application software respectively. However, shortcomings of the language may prevent it from being applied to larger computers where its space-saving features are less useful. But it will continue to find many applications in small and interactive systems and real-time applications including hardware simulation, video games and test-equipment control.

References

8. M. McNeil, A hashed dictionary search method, FORML '81 papers, FIG.
9. T. Dowling, Hash encoded Forth Fields, FORM '81 papers, FIG.
10. K. Schleisiek, Separated heads, *Forth Dimensions*, vol. 12, no 5, pp. 147-150, FIG.
11. D. Petty, Utilizing vocabularies, FORML '81 papers, FIG.
12. J. Cassady, Towards a 79-Standard Fig-Forth Romable Vocabulary and Smart FORGET, FORML '81 papers, FIG.
13. G. Shaw, Executable Vocabulary Stack, FORML '81 papers, FIG.
14. J. Gilbreath, A. high-level Benchmark, *Byte*, Sep. 1981, pp.180-196 (for algorithm and performance data see *Byte*, Jan. 1983).
15. M. Jesch, High-level floating point, Forth Dimensions, vol.IV, no. 1, pp.6-12.