INTERACTIVE

JANUARY, 1982 ISSUE NO. 7

SPECIAL FORTH (mostly) ISSUE

3 Introduction to Forth
4 Countdown Timer
5 Errata and Update to Forth Manual
7 Centronics—Type Printer Driver
8 Uncolon . . . A Forth Disassembler
9 AIM 65 ROM Expansion
10 2716/2532 EPROM Programmer
19 Frequency Counter
21 Random Number Generator
21 System Spotlight
23 Using Smudge
23 Number Input Function
24 Dumping Basic
24 Cassette Problem

Rockwell International

where science gets down to business

EDITOR'S CORNER

INTERACTIVE NOW FREE!!!

That's right. We've decided to drop the subscription price for the newsletter. So there is no need to send more money—just articles! However, since this newsletter is being mailed bulk rate, you'll have to continue to keep us up to date on your current mailing address. Send that information to: Subscription Desk, Mail Stop RC55, Rockwell International POB 3669, Anaheim, CA 92803. You don't need to re-register. If you received this issue through the mail, you're on the list and will continue to receive INTERACTIVE at no charge. If you're not on the list, send your name and address to our subscription dept.

RM65 BOARD STATUS

As I write this (December 4, 1981) all boards are available from stock except for the CRT Controller and the Floppy Disk Controller modules. These two boards are expected to be ready by the end of this month, however.

How many of you would like to see INTERACTIVE publish a monitor program for the RM65 Single-Board Computer module so it could be used standalone with the ACIA board as a complete system?

If you're looking for a compact, easy to package, 6502 system, you could not find one much better than the RM65 module set. The only problem at this time is that there is no software available to "tie it all together." I'd be willing to take on the task of developing a mini-monitor for the SBC if there is enough interest. How about it?

SPECIAL FORTH ISSUE

No, this isn't the first publication to dedicate an issue to FORTH. But FORTH is such an important addition to our language "arsenal" that I don't see how it could get too much coverage. To help seed the field, I even sat down and wrote the EPROM programmer driver software in FORTH (somewhere in this issue) to give you an idea of what FORTH is all about.

If you're a newcomer to FORTH, I strongly suggest that you purchase a copy of the AIM 65 FORTH USER MANUAL from your local Rockwell sales office. Another book that is very good is called STARTING FORTH. It is written by Leo Brodie and is available from Mountain View Press, POB 4656, Mountain View, CA 94040 (415-961-4103). The book costs around $16 (here in the states) and is an excellent companion to the AIM 65 FORTH manual.

ISSUES #1 AND #2 ALL GONE

We are completely out of stock of these two issues and have no plans on reprinting in the near future.

CORRECTIONS TO ISSUE #6

PAGE 3 —in the second paragraph on the right hand side of the page, change the last part of the third sentence to read ". . . size to 3686." Also, the cursor positions indicated came out in the wrong positions.

PAGE 21—pin 21 of U2 (the 2716 EPROM) must be connected to +5 volts.

ERIC C. REHNKE—EDITOR

All subscription correspondence and articles should be sent to:

EDITOR, INTERACTIVE
ROCKWELL INTERNATIONAL
POB 3669, RC 55
ANAHEIM, CA 92803

COPYRIGHT 1982 ROCKWELL INTERNATIONAL CORPORATION

Rockwell does not assume any liability arising out of the application or use of any products, circuit, or software described herein, neither does it convey any license under its patent rights nor the patent rights of others. Rockwell further reserves the right to make changes in any products herein without notice.

FORTH IN MAINLAND CHINA

Add Forth to the list of Chinese dialects. Dr. Ray Dessy of the Virginia Polytechnic Institute made a visit to Beijing's Chinese Academy of Science, Institute of Chemistry, as part of an exchange program for laboratory automation. He took along his Rockwell AIM 65 microcomputer (programmable in Forth) to demonstrate points in his 21-hour lecture series.

The Forth machine remained behind while he made a side trip to Shanghai. While Dessy was gone, his Chinese colleagues put the microcomputer to work in their instrumentation laboratory. They were most impressed by the ease with which Forth can be tailored to special uses. And they only reluctantly gave up the hardware when he returned.

In addition to the academic interest, Forth is also hard at work in 17 petroleum-refining research pilot plants in China. The CBX/5 computer-control systems were supplied by the Xytel Corp. and use the polyFORTH software package from Forth, Inc. The Research Institute of Petroleum Processing, at Beijing, was so pleased with polyFORTH's performance that it signed up for a system-wide software license.

FORTH TIDBIT

If you get an error when you attempt to compile a FORTH program from the AIM 65 text editor, location $00DF contains the address of the text pointer where the error occurred. To find that area, simply execute HEX DF @ CR 20 − 20 TYPE to print the 20 (hex) characters in front of the place where the error occurred.

INTRODUCTION TO FORTH

Gordon Smith
Rockwell International

FORTH, is an extensible language. If you need a capability or instruction that the language implementers did not include, you may add it yourself. In fact, programming in FORTH is accomplished by adding extensions to its command set until the final extension is the command to execute the final program. There are no reserved words or characters (except space and backspace) that the programmer cannot use for FORTH command names. If the programmer wishes, even existing FORTH words may be redefined and the compiler operation modified.
It is a procedure-oriented language. To call a procedure all the programmer does is use the procedure name. This may be included as part of the definition of a new procedure or it may be used interactively under keyboard control. Just typing the name of the procedure followed by a carriage return causes that procedure to execute.

FORTH is extremely easy to debug since immediately after defining a new procedure (called a word in FORTH), the procedure may be checked without any disk or editor or linker or additional programming manipulation. Just put some data where the procedure expects it, call the procedure, and see if the results were as predicted. As each new step of the program is written, it may be checked immediately. Any debugging required only relates to the single procedure just completed. FORTH is the most easily debugged of all the higher-level languages.

FORTH is a stack-oriented language. The stack (also called parameter stack) resembles the hardware stack in that data may be put onto the stack and removed from the stack without any address referencing. Most FORTH instructions expect the data for each procedure to be on top of the stack and they also leave the results of the procedure on the stack although access to any location in memory is allowed. To put data into the stack, just type the values desired (separated by spaces) and key the carriage return. After calling the procedure by typing its name, the results may be inspected by typing a print command (.). This causes the top value on the stack to be printed and deleted from the stack. The stack orientation also eliminates some of the debugging problems that assembly language and the BASICs have. What would be termed local variables in PASCAL (variables associated with a procedure and not the global variables associated with the whole program) are contained in the stack and in most cases do not even have names. Consequently, the errors associated with reusing a name or referencing the wrong data name are unlikely with FORTH.

FORTH is a structured language. There is no GOTO command to cause complicated intertwining of the program. It does have all of the normal structured programming formats:

```
IF ..................THEN
IF ..................ELSE ................THEN
BEGIN ..............WHILE...............REPEAT
BEGIN ..............UNTIL
BEGIN ...............AGAIN
DO .................LOOP
DO ................+LOOP
```

FORTH has elements of a compiler as well as an interpreter in it. It also has an assembly language capability built into it so that high speed is possible. Normal high level FORTH is about the same speed as PASCAL but since it has the direct assembly language capability as well, it can have the most inner loop procedures defined in machine language so that it can handle most real time control functions readily.

FORTH is also extremely compact. The FORTH program ROMs that Rockwell offers with the RM 65 board module line and the AIM 65 and AIM 65/40 single board computers provide the compiler, the interpreter, the assembler, a mass storage operating system and a total of approximately 300 procedures all in a total of 8K bytes. An applications program is obtained by programming additional procedures until the final procedure is the complete program the user desires. These procedures may also be implemented in ROM or PROM. It is possible to compress the program even more by generating new ROMs or PROMs which contain only the FORTH procedures actually used in the application and which omit the name information that is no longer needed. A FORTH application program, including the overhead, can be even shorter than if the program were completely written in assembly language!

Finally FORTH gives the user much more intimate control of the computer than any of the other languages. The FORTH programmer may switch from the normal decimal operation to hexadecimal for addressing memory mapped input/output locations, to binary for manipulating bit flags for control or status information, and to base 36 arithmetic (using the character 0-9, A-Z for the 36 symbols required) for compact storage of text for prompts and messages. The programmer may directly address any location in memory for any purpose.

FORTH is a continually growing language because of its extensibility and flexibility, but it is highly standardized. There is an international standards organization which has defined a minimum core set of FORTH procedures (called "words" in FORTH) and has defined a proposed set of extension words so that FORTH programs will continue to be highly transportable from one computer to another. The AIM 65 FORTH was defined before the standard was published, but it does meet most of the requirements and does, in fact, include many additional features.

In most computer implementations of FORTH, it is RAM based. Rockwell, in order to have modular software to go with its line of modular boards and computers, has implemented FORTH in ROM. This makes it particularly applicable to industrial control where both the run-time package and the application package may be in non-volatile memory.

The Rockwell AIM 65 computer makes an efficient, low cost, development system for developing and debugging FORTH programs. A resulting program then may be implemented in PROM and used in either the AIM 65 packaged into the user application or may be used in a computer made from RM 65 modules.

COUNTDOWN TIMER

Randy Dumse
Rockwell International

The program shown uses the R6522 on the AIM 65 system as a time base for a count down timer. It is written in FORTH and demonstrates the flexibility of the language and its usefulness to AIM 65 system users. Once the program has been compiled a user has only to type the number of seconds delay that is desired, followed by a space and the word SECONDS, then hit return. The seconds count will decrement on the display and print OK when the time is up.

A look at the words as they are defined will reveal this program's inner workings. The word SECONDS first puts a zero on the stack above the user entered value for seconds, then the address of the variable HSEC and does a store operation (!). This zeros HSEC. The number of seconds to be counted is again at the top of the stack. The address of the variable SEC is added to the stack so the store operation (!) can place the value of seconds into that variable.

The word START-TIMER is executed which starts the R6522 Timer 1 in the free running mode. Timer 1 will time out every 1/20 of a second from that point on. A loop is set up to be the word BEGIN that will continue until the variable SEC becomes zero. The word HSEC-TICK waits for Timer 1 to time out then returns. The word UPDATE adds 5 to the hundreths of a second count and updates the second count after a full second has passed.

Following that the word DISPLAY checks to see if the hundreth of a second count is zero. If so, it clears the display and prints the newly updated second count. The address of SEC is then put on the stack so the fetch command (@) can bring its value to the stack. The value is tested by the word NOT and the loop begun again by UNTIL if the second count is not yet zero.

The word DISPLAY operates by putting the address of HSEC on the stack and using the fetch command (@) to get its value which is then inverted by NOT. IF skips the commands following all the way to the word THEN if HSEC did not contain a zero. If it did however the machine code routine CLR is called, the address of SEC is used to fetch (@) the seconds count and that value is printed (.).

The word UPDATE performs its function by first getting the current hundreth second count (HSEC @) and then adding five to the count with the words 5+. The result is duplicated (DUP) to allow a copy for testing purposes. A 99 is put on the stack and compared to this duplicate copy by the > operator. If the result of the test is false execution will skip from the IF word to the THEN. If the result is true however the word DO-SEC is executed and the hundreth second count dropped from the stack (DROP) and replaced by a zero. After THEN, which ever value remains on the stack is stored in HSEC (HSEC !).

DO-SEC is a fairly simple word that fetches the value from second count variable (SEC @), subtracts one from it (1−) and returns this value back (SEC !).

The word CLR takes advantage of the ease with which FORTH can decend to assembly language (machine coded) routines. Because there is a very useful routine in the monitor that clears the display but doesn't run the printer (called CLR in the monitor listing, address $EB44), the word CLR in this program is created with the defining word CODE. The two machine level instructions call the monitor routine (EB44 JSR,) and return control back to FORTH (NEXT JMP,). END-CODE terminates this word.

The word HSEC-TICK is defined simply as calling the two other words WAIT-ON-TIMER and CLEAR-FLAG. WAIT-ON-TIMER starts a loop with the word BEGIN that continues until Timer 1 sets its flag by timing out. The Flag Register is fetched by the words 6522-TIMER-FLAGS and C@. The C@ insures that only one register is read from the R6522. Reading more than one register could give invalid results. This copy of the Flag Register is logically AND'ed with $40 so that only the T1 flag value remains. The word UNTIL repeats the loop until the T1 flag is no longer zero.

The timer is started by the word START-TIMER. A $CØ is put in its auxiliary control register (CØ ACR C!) and a value of 50,000 in its counter (HSEC-COUNT-VALUE 6522-TIMER !).

The timer flag is cleared by the word CLEAR-FLAG each time it is set. CLEAR-FLAG simply reads the low byte of the counter (6522-TIMER C@) which causes the flag in the Flag Register to clear and throws away the fetched value (DROP) which has no use in the program. The remaining listing lines not described above, simply define the variables and constants used elsewhere in the program.

If the user wants more than the OK prompt to indicate the end of the count down, another word could be added. It would use SECONDS as shown and end with a rountine such as CR.

When SECONDS was done the CR would run a line on the printer, giving an audible indication of completion. (i.e., :ALARM SECONDS CR ;). If something fancy is required a user defined word could replace CR and sound a buzzer, ring a bell or flash a light.

```
( COUNTDOWN TIMER PROGRAM BY R. DUMSE)
HEX

A004 CONSTANT 6522-TIMER
A00D CONSTANT 6522-TIMER-FLAGS
A00B CONSTANT ACR

DECIMAL

50000 2 - CONSTANT HSEC-COUNT-VALUE
HEX
0 VARIABLE HSEC
0 VARIABLE SEC

: WAIT-ON-TIMER  BEGIN
                   6522-TIMER-FLAGS
                   C@ 40 AND
                  UNTIL  ;

: START-TIMER  C0 ACR C!
               HSEC-COUNT-VALUE
               6522-TIMER !  ;

: CLEAR-FLAG  6522-TIMER C@ DROP ;

: HSEC-TICK  WAIT-ON-TIMER CLEAR-FLAG ;

CODE  CLR
   EB44 JSR,
   NEXT JMP,
END-CODE

DECIMAL
: DO-SEC  SEC @ 1- SEC ! ;

: UPDATE  HSEC @ 5 + DUP
          99 >
           IF
            DO-SEC DROP 0
           THEN
          HSEC ! ;

: DISPLAY  HSEC @ NOT
            IF
             CLR SEC @ .
            THEN  ;

: SECONDS  0 HSEC ! SEC ! START-TIMER
           BEGIN
            HSEC-TICK UPDATE DISPLAY
            SEC @ NOT
           UNTIL  ;

FINIS
```

ERRATA AND UPDATE TO FORTH MANUAL

p. 4-30 — If you get a compilation error when compiling from the text editor, the 16 bit pointer at location $00DF will indicate where the error occurred in the text buffer.

Perform a

HEX DF @ 20 – CR 20 TYPE

to see the last 20 characters before the error occurred.

p. 4-31 — The last sentence in the warning should start "FORGETing TASK then . . ."

p. 4-37 — The explanation for LEAVE is incomplete. Refer to the definition on page B-29.

p. 4-55 — The short sequence at the top of the page SHOULD read

BASE @ DUP DECIMAL.

The word BASE? should be defined as follows:

: BASE? BASE @ DUP DECIMAL .
BASE ! ;

Otherwise, BASE gets changed to decimal whenever BASE? is executed.

p. 4-57 — The sequence at the top of the page should read:

DECIMAL 65 EMIT

p. 5-1 — The second paragraph in section 5.1.1 should read:

". . . quotient (top of the stack) and the remainder (second on the stack) . . .

p. 5-14 — Delete the two numbers (500 5) from the example at the bottom of the page. Those numbers will be on the stack from the sequence immediately above. You must press the RETURN key after CR TYPE CR.

p. 5-19 — Change the number 7F to 1F in the second code sequence from the bottom of the page. The last code sequence should read:

LATEST CR ID.

p. 6-6 — The hex data at location $030F should be changed from 04 to D4.

more ——▸

p. 6-11 — RP) does not work correctly. It returns 101 decimal (65 hex) instead of 101 hex. If you need to use it in your assembly language you'll have to redefine it.

HEX 101 CONSTANT RP)

p. 6-13 — The text in the parenthesis at the bottom of the page should read:

(the <space> bar was pressed here)

p. 7-4 — Delete the second line (ASSEMBLER) in the example in 7.2.1. The assembler will be called when CODE is executed.

p. 7-10 — In 7.3.4d the code should read:

: DUMMY ARM [SMUDGE

see J-8 for ARM.

p. 8-6 — Don't forget to insert a space immediately after the first parenthesis in all the comment lines in both program examples.

p. 8-10 — Again the comment in the ?STROBE example needs to have a space after the first parenthesis.

p. 10-5 — The first sentence in paragraph 9 should start "Set and verify . . ."

p. 10-6 — Change step 12 to read:

(12) Restore dummy word TASK, verify that TASK is entered into the FORTH dictionary and its PFA is $309 and read the dictionary pointer (DP).

: TASK ;
VLIST
309 TASK <space>

DP @ . <return> 30B OK

Note that any new words now added to the dictionary will be located at $30B on up.

p. 10-7 — Change step 13 also.

(13) Display the link field address of TASK to get the address of the last application word for use in step 15.

' TASK LFA .S @ 0 D.

<return> 305 LAST

p. 10-7 — Change the assembly language instruction in Section 15 from LDA A0,Y to LDA (A4),Y

p. 10-11 — At the top of the page change * = 800 to * = D000.

Appendix B-41 — Remove 'addr' from the stack notation of 'WORD'. Also delete the last sentence in the definition.

Appendix G-3 — The number of bytes for parameter name MODE is 2 (not 8).

Appendix I-2 — 255 places should be allotted to the variable IB (not 25).

0 VARIABLE IB
255 ALLOT

Appendix I — Forth String Words.

To make this string package more nearly compatible with the one found in BASIC several changes are necessary.

Change the definitions for MID$, LEFT$ and RIGHT$ to:

: MID$ DROP 1- ROT + SWAP ;
: LEFT$ DROP SWAP ;
: RIGHT$ DUP 4 PICK MIN - + SWAP ;

Appendix I-6 — Under explanation for S!, there should be a space inserted between the double quotes and 'COWS'

" COWS not "COWS

change the explanation of MID$ to the following:

MID$ gets M characters of a string starting at the Nth character position, for example

6 3 A$ MID$ TYPE

will print the word EAT.

Appendix I-7 — Under the explanation for VAL delete the space following the number 128.

" 128" VAL D.

Appendix I-8 — Under the explanation for SUB the second program line should read:

" ATE" 6 3 A$ MID$ SUB

Appendix K-1 — In the definition for OFF, the word DROP right before the semicolon should be deleted. It should read:

: OFF A004 @ 12B + DUP CR IF FFFF DNEGATE D. ELSE . THEN ;

CENTRONICS-TYPE PRINTER DRIVER

Joe Hance
Rockwell International

(EDITOR'S NOTE—Joe has an EPSON MX 80 hooked up to his AIM 65 and is mighty happy with it. Here's how he did it.)

```
2000                    ; THE HARDWARE CONNECTIONS ARE :
2000                    ;
2000                    ; CB2 ---> STROBE
2000                    ; CB1 ---> ACK
2000                    ; PB0 ---> DAT0
2000                    ; PB1 ---> DAT1
2000                    ; PB2 ---> DAT2
2000                    ; PB3 ---> DAT3
2000                    ; PB4 ---> DAT4
2000                    ; PB5 ---> DAT5
2000                    ; PB6 ---> DAT6
2000                    ; PB7 ---> DAT7
2000                    ; GND ---> GND
2000                    ;
2000                    PB      =$A000          ; PORT B OF R6522
2000                    DPB     =$A002          ; PORT B DIRECTION
2000                    PCR     =$A00C          ; CONTROL REGISTER
2000                    IFR     =$A00D          ; FLAG REGISTER
2000                    ;
2000                    ;
2000                    ; USER OUTPUT VECTOR
2000                    ; COMES HERE
2000                    ;
2000                            *=$0FE0
0FE0
0FE0  90 0C             UOUT    BCC INIT        ; CARRY CLEAR IF FIRST TIME
0FE2  A9 10                     LDA #$10        ; LOOK FOR CB1 TRANSITION
0FE4  2C 0D A0          WAIT    BIT IFR         ; WAIT UNTIL FLAG
0FE7  F0 FB                     BEQ WAIT
0FE9  68                        PLA             ; GET CHARACTER
0FEA  8D 00 A0                  STA PB          ; STORE TO PORT
0FED  60                        RTS
0FEE                    ;
0FEE                    ; INITIALIZE INTERFACE
0FEE                    ;
0FEE  A9 FF             INIT    LDA #$FF        ; PORT B TO OUTPUT
0FF0  8D 02 A0                  STA DPB
0FF3  A9 A0                     LDA #$A0        ; CB2 HANDSHAKE AN
0FF5  8D 0C A0                  STA PCR         ;   CB1+TRANSITION
0FF8  60                        RTS
0FF9                            .END
```

UNCOLON ... A FORTH DISASSEMBLER

Once a FORTH word has been compiled, you can't normally list it unless you've saved the source code. It's sort of like assembly language in that respect. Of course, once a program has been converted to machine code, you could use a disassemble to convert the opcodes back into their mnemonic form to get some idea of what's going on.

Well, here's a disassembler (if we can use the term correctly) for FORTH words. Actually, the word UNCOLON (de-Forth?) seems to fit the operation more closely.

What it does is decompile higlevel FORTH words—(not CODE words) into the subwords that make them up.

The original author for this program is unknown. Gordon Smith, here at Rockwell, modified the program somewhat and is presenting it to us to add to our FORTH toolbox. UNCOLON has proven to be very handy.

Once you have entered it into your system, use it as follows:To decompile .R execute the following:

' .R UN:

What you are doing is entering UN: with the parameter field address (PFA) of the word .R on the stack.

If you typed in all the definitions correctly, you'll get the following printout:

```
C959    B589    >R
C95B    C311    S->D
C95D    B599    R>
C95F    C925    D.R
C961    B55B    ;S OK
```

Here is what a DO . . . LOOP looks like when decompiled:

First the definition:

: TEST 1000 0 DO I . LOOP ;

Now, decompile TEST.

' TEST UN: <CR>

```
5A4     B040    LIT
5A6     1000
5A8     B7F1    Ø
5AA     B173    (DO)
5AC     B18C    I
5AE     C967
5BO     B10C    (LOOP) 5AC

5B4     B55B    ;S OK
```

Experiment with other FORTH constructs such as BEGIN . . . UNTIL, IF . . . ELSE . . . THEN etc. to familiarize yourself with the decompiler.

As you will notice when you compare the above disassembly with what you wrote, sometimes FORTH does some processing while it compiles a word. The 1000 that was written is repeated in the definition but after an internal FORTH word called LIT (LIT tells FORTH that the next word encountered is really a number and not an address). The 0 (zero) which was the next entry in the definition was not written as a LIT sequence. This is because 0 (zero) is defined in FORTH as a CONSTANT with the name 0 (zero).

The DO word is an immediate word in FORTH which executes even though FORTH is in the compiling mode. What DO actually does is to compile (DO) and put the address of the first command in the loop on the stack. The remaining commands until the LOOP word are compiled directly. LOOP, however, is another immediate word which compiles (LOOP) into the dictionary and computes the relative address (16 bits) to get back to the address which was placed on the stack by DO.

The disassembler shows the absolute address for convenience, but it was the 16 bit relative address which was compiled to the dictionary. The ; (semicolon) gets compiled as ;S which tells the interpreter to get the next command. The (DO) command when it is executed takes the two top values off the stack and puts them in the return stack. The (LOOP) command pulls the top two values off the return stack, increments the index and if the index is not equal to or greater than the limit value, it puts the new values back on the return stack and does a 16 bit relative branch back to the beginning of the DO loop. If the results of the incrementing indicate termination of the loop, the two values are dropped and the relative address is ignored.

As you can see, you can learn alot about how FORTH operates by using the UN: command.

The decompiler doesn't handle CODE defined or other special cases words but does a good job with most others.

Now you can decompile FORTH to see how it works and to become more familiar with how the experts use it.

```
( DISASSEMBLE )

HEX
: EXIT R> R> DROP DROP ;

: IP&W ( CFA --- CFA
  CR DUP DUP 0 4
  D.R @ DUP 0 5 D.R
  SPACE ;

: WRITELN IP&W DUP
  ' CLIT CFA =
  IF ." CLIT "
   DROP 2+ DUP C@
   0 D. 1-
  ELSE 2+ NFA ID.
  THEN ;

: ?LIT  ( PFA---PFA F )
  DUP @ ' LIT CFA
  = ;

: ?CPLE ( PFA---PFA F )
  DUP @ ' COMPILE
  CFA = ;

: ?STOP ( PFA---PFA F )
  DUP @ DUP ' ;S CFA =
  OVER ' (;CODE)
  CFA = OR SWAP
  DROP ;

: ?BRA DUP @ DUP
  B10C = OVER
  B13C =
  OR OVER B0EB
  = OR OVER
  B0CC = OR
  SWAP DROP ;

: ?STRING ( PFA---PFA F )
  DUP @ ' (.") CFA
  = ;

: STRNG ( PFA---PFA+2 )
  2+ DUP COUNT DUP
  >R TYPE R> 1- + ;

: DISASM ( PFA--- )
  BEGIN
   ?TERMINAL IF
  EXIT THEN
    WRITELN ?STOP
    0=
```

AIM-65 ROM EXPANSION

I just received a flyer from a company that says they have a ROM expansion board for AIM-65 computers. It looks like a bare p.c. board that can hold up to 6 ROM/EPROMs and a selector for the chip select lines. The board mounts over top of the ROM sockets and connects to the computer with two 24-pin dip headers. They're asking $12.50 for the bare p.c. board.

You can contact them at:

MICRO PROCESSOR PRODUCTS
2916 EAST COURT ST.
IOWA CITY, IOWA 52240
(319) 351-7587

```
   WHILE
    ?LIT IF 2+ IP&W
     DROP ELSE
    ?BRA IF 2+ DUP @
     OVER + 0 D.
     ELSE
    ?CPLE IF 2+ DUP
     @ 2+ NFA ID.
     ELSE
    ?STRING IF STRNG
    THEN THEN THEN
    THEN 2+
   REPEAT
    DROP ;

: UN: BASE @ SWAP
   HEX DUP CFA @ DUP
   B756 = IF DROP DISASM
   ELSE DUP B7AE = IF
    DROP DROP
    CR ." VARIABLE "
   ELSE DUP B792 = IF
    DROP DROP
    CR ." CONSTANT "
   ELSE DUP B7DE =
    IF DROP DROP
     CR ." USER VARIABLE "
   ELSE BC79 = IF
    DROP CR ." DEFINED WORD "
   ELSE CR ." CODE OR SPECIAL @ "
    0 D. THEN THEN THEN
   THEN THEN BASE ! ;
```

FINIS

2716/2532 EPROM PROGRAMMER

Eric C. Rehnke
Editor

Here's an EPROM programmer (hardware and software) that will burn the two most popular 5 volt only devices (2716 and 2532). The software driver is written in FORTH to give you an idea of how this new language can be applied to "real world" applications.

As you can see, the hardware design is fairly simple and straightforward. It can be operated from the user R6522 VIA or an AIM 65, AIM 65/40, or RM 65 single board computers. The DC-DC converter (VA-15 15) and output regulator (723) are optional and can be replaced by a 25 volt supply if one is available.

Operation of the unit is also fairly straightforward.
A CMOS 4040 12-bit counter chip generates the addresses for the EPROM. This significantly cuts down on the number of I/O lines needed for the interface by giving two lines the power of 12.

Reed relays are used to control the two voltages that are needed (+5 and +25). These relays are driven through two open collector buffers (part of the 7407) and a DPDT switch is used to accomodate the slight differences between the Intel (2K) 2716 and the TI 2532 (4K) pinouts.

The connection from pin 20 of the EPROM socket to PB7 of the R6522 is used along with a bit of software to let us read the position of the selector switch.

Notice how the program was built up from low-level words (routines) such as READ-PORTB and WRITE-PORTB which evolved into words such as CS=HIGH, PULSE=READ and 25V-ON.

See how FORTH programs grow?

The software driver was written to use names as descriptive as possible to increase readability and make the definitions easy to follow. The program compiles from the AIM 65 system text editor which occupies about 8K bytes. The compiled object code occupies slightly less than 2K bytes however.

There are two ways to command the unit to program an EPROM. You can execute PROGRAM, whereby the software will prompt you for the start and end addresses of the data to be burned into EPROM as well as the starting address of the data in the EPROM and the EPROM type (2716 or 2532).

Or, you can execute BURN-EPROM with the starting address and ending address of the data to get burned into the EPROM as well as the target address of the data in the EPROM on the stack and the switch in the proper position.

Suppose you had 256 bytes of data located at $2100 and you wanted it to be located one page up from the start of the EPROM. You would execute the phrase

HEX 2100 21FF 0100 BURN-EPROM

to use the quick program entry.

The program would then automatically make sure the EPROM is erased in the area that you will be programming and verify that it had programmed correctly after it completed the programming cycle. Messages are displayed during these three operations so you know what's going on.

When executing BURN-EPROM, there are no checks made for addresses out of range or the switch being in the wrong position.

These "idiot checks" are made, however, when you enter by executing PROGRAM (nothing need be on the stack).

The first prompt you'll receive when you execute PROGRAM will be

START ADDRESS=

Type in the starting address (in hex) of the memory location where the data to be programmed into the EPROM is located. Then press the RETURN key (this will terminate all the inputs). By the way, the printer will be turned on here to record the prompts and the parameters that you type in.

The next prompt to appear is

END ADDRESS=

Type in the last address of the memory locations where the data to be programmed into the EPROM is located. Terminate with a carriage return.

TARGET ADDRESS=

is the next prompt to appear and refers to the address of the first location IN THE EPROM where the data is to be programmed.

You can program any number of locations (even just one) starting at any address within the EPROM.

If the data is to be burned into the EPROM starting at the first location in the EPROM you would enter 0000 for the target address (or just 0). However, if the data was to start at the 80th (decimal) location in the EPROM, you would enter 50 (hex) as the target address.

The ADJ-EPROM-START routine ignores the top 4 or 5 bits of the target address (depending on whether the EPROM is a 2K or 4K size) so the target address could be entered as the actual memory address of the EPROM when it is installed in the system.

So, if the EPROM were to reside at $2000 in the system and you wanted the data to be located one page up into the EPROM, you could type in 2100 or, 100 since the address gets "corrected" by ANDing it with the appropriate mask.

The last user prompt is

EPROM TYPE=

You're expected to enter the correct EPROM type number here (either 2716 or 2532). The type number you enter must also agree with the setting of the selection switch. If there are any discrepancies, the type prompt will reappear.

If you entered the correct number, the program will turn off the printer and start checking the EPROM to make sure that the area you wish to program has been erased (each location should contain FF). The display will contain the message NOW CHECKING.

After that the actual programming will commence signified by the message NOW PROGRAMMING on the display.

Finally, the program will go back and verify that the EPROM was programmed correctly. NOW VERIFYING will be displayed. Any differences will cause the program to abort and print out the first address that didn't get programmed correctly.

These two interfaces are included just to give you an idea of what can be done with FORTH. There is more error checking when you start at PROGRAM. The end address is tested to be greater than the start address but not by more than the EPROM capacity.

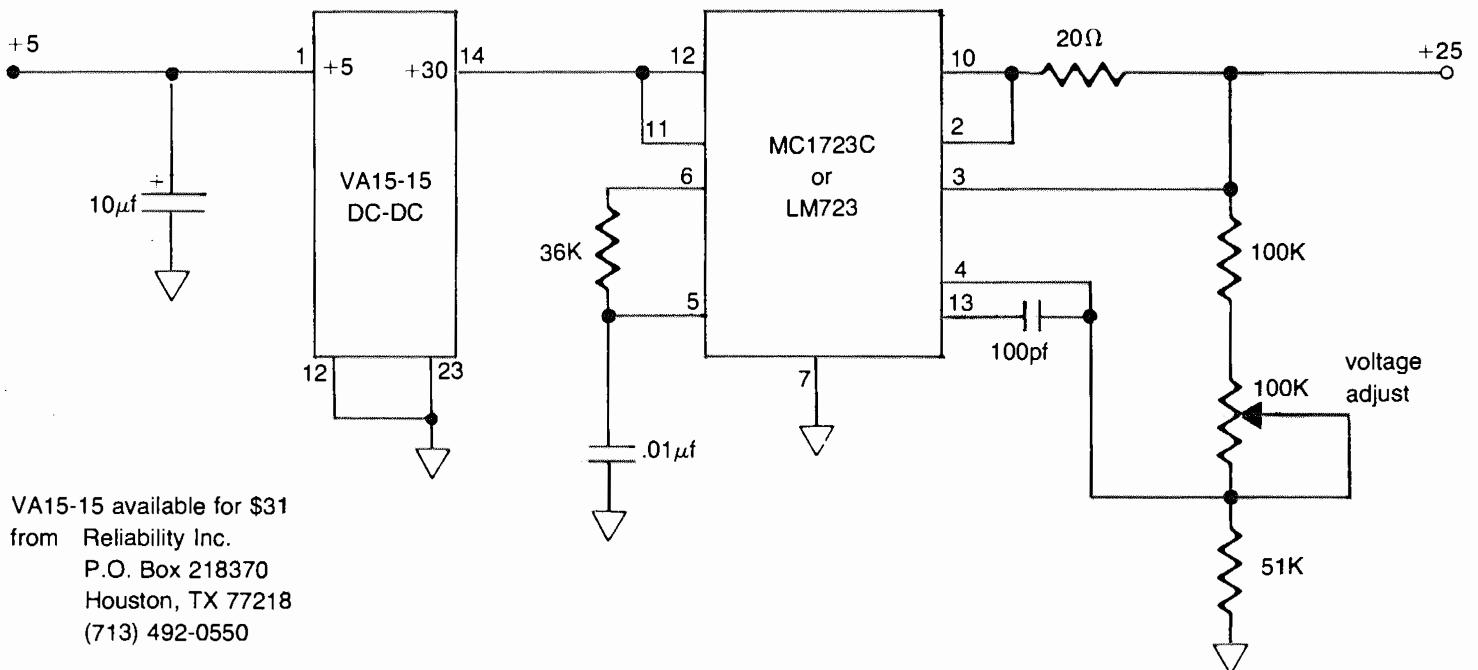
There is also a utility included called READ-EPROM. It reads the contents of an EPROM into a specified memory buffer area.

The size of the buffer used (2K or 4K) will depend on the switch setting. If you wanted to read a 2532 EPROM into memory starting at $2000, you would set the switch to 2532 and execute the phrase

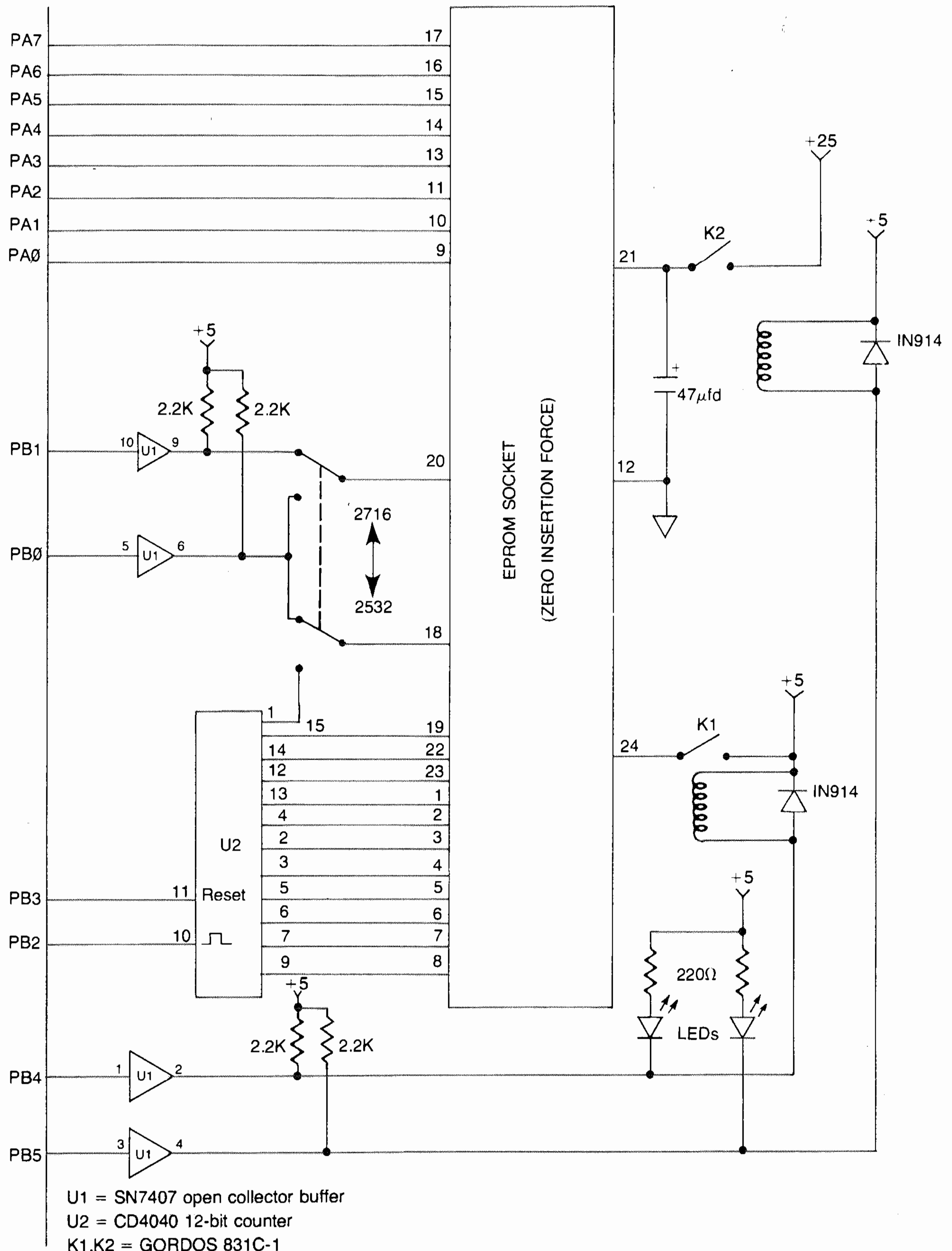
HEX 2000 READ-EPROM

Hopefully, this program will provide you with some programming examples to study and perhaps some words you can pull out and use in one of your own programs.

That's a good way to learn this language. FORTH seems odd at first, but give yourself some time and you'll begin to master it.



OPTIONAL POWER SUPPLY



EPROM PROGRAMMER

```
( EPROM PROGRAMMING SOFTWARE  --ECR 10/14/81)
HEX

( 6522 CONSTANTS FOLLOW)

A000 CONSTANT PORTB
A002 CONSTANT PBDD
A00D CONSTANT IFR
A008 CONSTANT T2L
A009 CONSTANT T2H
A00B CONSTANT ACR
A00F CONSTANT PORTA
A003 CONSTANT PADD

( ROMABLE VARIABLES FOLLOW)

60 USER EPROM-TYPE SMUDGE
62 USER RAM-START SMUDGE
64 USER RAM-END SMUDGE
66 USER EPROM-START SMUDGE
68 USER COUNTER SMUDGE

CODE PULSE-IT    ( OUTPUT A 50 MS PULSE)
                 ( ON PULSE LINE -PB0   )

( FIRST INVERT THE SIGNAL AT PB0)
PORTB LDA,  1 # EOR,  PORTB STA,
( THEN STARTUP THE TIMER)
0 # LDA,  ACR STA,  T2L STA,
C3 # LDA,  T2H STA,
20 # LDA,
BEGIN,   ( LOOP TIL TIME-OUT)
  IFR BIT,
  0= NOT
UNTIL,
( INVERT PB0 AGAIN)
PORTB LDA,  1 # EOR,  PORTB STA,
T2L LDA,  ( CLEAR FLAG)
NEXT JMP,
END-CODE


: EPROM-TYPE?  EPROM-TYPE @ ;

: PORTA=INPUT  0 PADD C! ;

: PORTA=OUTPUT  FF PADD C! ;

: WRITE-PORTA  PORTA C! ;

: READ-PORTA  PORTA C@ ;

: WRITE-PORTB  PORTB C! ;

: READ-PORTB  PORTB C@ ;
```

```
: BUMP-COUNTER   ( INCREMENT ADDRESS)
                 ( COUNTER BY ONE    )

   READ-PORTB 04 OR DUP WRITE-PORTB
   FB AND WRITE-PORTB 1 COUNTER +! ;

: SETUP-PORTB   ( INITIALIZE PORTB SO)
                ( PB0-PB6 ARE OUTPUTS)
                ( AND PB7 IS AN INPUT)

   FF WRITE-PORTB 7F PBDD C! ;

: RESET-COUNTER   ( RESET ADDRESS     )
                  ( COUNTER TO ZERO   )

   READ-PORTB 08 OR DUP WRITE-PORTB
   F7 AND WRITE-PORTB 0 COUNTER ! ;

: BURN-BYTE   ( N --> ---                     )
              ( WRITE LOWER EIGHT BITS OF N   )
              ( INTO EPROM AT PRESENT ADDRESS)
              ( COUNTER LOCATION              )

   WRITE-PORTA PULSE-IT BUMP-COUNTER ;

( THE FOLLOWING FOUR WORDS TURN THE)
( APPROPRIATE REED RELAYS ON OR OFF)
( WHICH SUPPLY VOLTAGE TO THE EPROM)

: 25V-OFF  READ-PORTB 20 OR WRITE-PORTB ;

: 25V-ON  READ-PORTB DF AND WRITE-PORTB ;

: 5V-OFF  READ-PORTB 10 OR WRITE-PORTB ;

: 5V-ON  READ-PORTB EF AND WRITE-PORTB ;

: H-ON  ( TURN PRINTER ON)

   80 A411 C! ;

: H-OFF  ( TURN PRINTER OFF)

   0 A411 C! ;

( THE FOLLOWING THREE WORDS SET THE)
( STATE OF THE CS AND PULSE LINES  )

: CS=HIGH  READ-PORTB 02 OR WRITE-PORTB ;

: CS=LOW  READ-PORTB FD AND WRITE-PORTB ;

: PULSE=LOW  READ-PORTB FE AND WRITE-PORTB ;
```

```
: CHECK-TYPE   ( SET PULSE AND CS LINES AND)
               ( READ STATE OF EPROM SELECT)
               ( SWITCH ON PB7   )

     PULSE=LOW CS=HIGH
     READ-PORTB 80 AND 0= ;

: SET-PULSE-STATE    ( SET INITIAL POLARITY)
                     ( OF PULSE LINE ACCORDING)
                     ( TO EPROM TYPE   )

     READ-PORTB EPROM-TYPE?
     IF
        01 OR
     ELSE
        FE AND
     THEN
     WRITE-PORTB ;

: SET-EPROM-START    ( INCREMENT ADDRESS COUNTER)
                     ( UNTIL IT AGREES WITH THE )
                     ( EPROM TARGET ADDRESS      )

     RESET-COUNTER EPROM-START @
     BEGIN
        DUP 0= NOT
     WHILE
        BUMP-COUNTER 1-
     REPEAT
     DROP  ;

: ADJ-EPROM-START    ( CONVERTS THE TARGET ADDRESS)
                     ( TO AN ADDRESS INCREMENT FOR)
                     ( COUNTER AND THEN BUMPS UP   )
                     ( THE COUNTER TO THE PROPER   )
                     ( INITIAL VALUE               )

     EPROM-START @ EPROM-TYPE?
     IF
        FFF AND EPROM-START !
     ELSE
        7FF AND EPROM-START !
     THEN
     SET-EPROM-START ;

: BURN-LOOP    ( THIS ROUTINE DOES THE   )
               ( ACTUAL DIRTY WORK OF    )
               ( PROGRAMMING THE EPROM. )

     RAM-END @ 1+ RAM-START @
     DO
        I C@ BURN-BYTE
     LOOP ;
```

```
: BURN-IT        ( SET EVERYTHING UP FOR  )
                 ( BURN-LOOP.             )

   CR ." NOW PROGRAMMING"
   SET-PULSE-STATE RESET-COUNTER
   SET-EPROM-START CS=HIGH
   PORTA=OUTPUT 5V-ON 25V-ON
   BURN-LOOP 25V-OFF 5V-OFF ;

: VERIFY-LOOP    ( GO THROUGH THE PART THAT)
                 ( WAS PROGRAMMED AND MAKE )
                 ( THAT EVERYTHING WENT OK.)

   RAM-END @ 1+ RAM-START @
   DO
      READ-PORTA I C@ = 0=
      IF
         H-ON CR
         ." DIDN'T PROGRAM CORRECTLY AT $"
         EPROM-START @ I RAM-START @ - + .
         CR 5V-OFF ABORT
      THEN
      BUMP-COUNTER
   LOOP ;

: CHECK-CLR-LOOP    ( USED BY CHECK-CLEAR TO)
                    ( CYCLE THROUGH EPROM   )
                    ( AND MAKE SURE THAT THE)
                    ( AREA TO BE PROGRAMMED )
                    ( HAS BEEN ERASED       )

   PORTA=INPUT
   RAM-END @ RAM-START @ - 1+
   0 DO
      READ-PORTA FF = 0=
      IF
        H-ON CR
        ." ALREADY PROGRAMMED AT $"
        CR I EPROM-START @ + .
        CR 5V-OFF ABORT
      THEN
      BUMP-COUNTER
   LOOP ;

: CHECK-CLEAR    ( SETUP EVERYTHING SO THAT)
                 ( CHECK-CLR-LOOP CAN DO   )
                 ( ITS THING.              )

   CR ." NOW CHECKING" SETUP-PORTB
   5V-ON CHECK-TYPE EPROM-TYPE !
   PULSE=LOW CS=LOW RESET-COUNTER
   ADJ-EPROM-START
   PORTA=INPUT CHECK-CLR-LOOP 5V-OFF ;
```

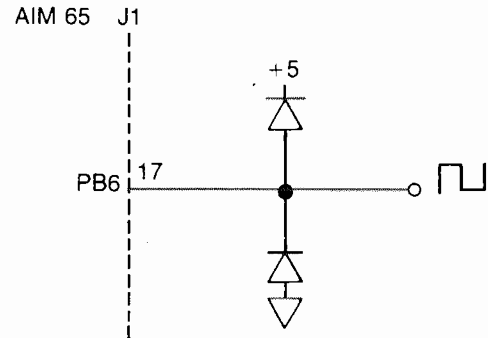
```
: VERIFY-CORRECT  ( SET IT ALL UP FOR VERIFY-LOOP)

    CR ." NOW VERIFYING  " SETUP-PORTB 5V-ON
    PORTA=INPUT PULSE=LOW CS=LOW RESET-COUNTER
    SET-EPROM-START VERIFY-LOOP 5V-OFF ;

: READ-EPROM  ( ADDR -->               )
              ( READ CONTENTS OF EPROM)
              ( INTO 2K OR 4K BUFFER   )
              ( STARTING AT ADDR       )

    SETUP-PORTB PORTA=INPUT 5V-ON
    RESET-COUNTER CHECK-TYPE EPROM-TYPE !
    CS=LOW PULSE=LOW EPROM-TYPE?
    IF
       1000
    ELSE
       800
    THEN
    0 DO
       DUP READ-PORTA SWAP C!
       1+ BUMP-COUNTER
    LOOP
    5V-OFF ;

: BURN-EPROM   ( THIS IS THE LOW LEVEL ENTRY )
               ( POINT FOR THE PROGRAMMING   )
               ( SOFTWARE. MAKE SURE THAT THE)
               ( START, END AND TARGET ADDRS )
               ( ARE ON THE STACK AND THE    )
               ( SELECT SWITCH IS CORRECTLY  )
               ( POSITIONED.                 )

    EPROM-START ! RAM-END ! RAM-START !
    CHECK-CLEAR BURN-IT VERIFY-CORRECT ;

: INPUT   ( GET A NUMBER FROM THE KEYBOARD)
          ( AND LEAVE IT AS A DOUBLE-     )
          ( PRECISION ON THE STACK        )

    PAD 10 EXPECT
    0 0 PAD 1- (NUMBER)
    DROP ;

: ERROR-EXIT   ( IN CASE OF AN ERROR, HERE'S A )
               ( CLEAN WAY TO LEAVE.           )

    H-ON CR ." ADDRESS ERROR" CR ABORT ;
```

```
: GET-FROM    ( GET THE FIRST ADDRESS OF THE DATA)
              ( THAT GETS BURNED INTO THE EPROM   )
              ( AND MAKE SURE IT IS A SINGLE-     )
              ( PRECISION NUMBER.  IF NOT, THEN   )
              ( LEAVE VIA ERROR-EXIT.             )

   CR ." START OF DATA=" INPUT 0=
   IF
      RAM-START !
   ELSE
      ERROR-EXIT
   THEN ;

( THE NEXT TWO DEFINITIONS ARE JUST ABOUT IDENTICAL)
( TO GET-FROM THAT THEY DON'T REALLY NEED TO BE     )
( EXPANDED OR EXPLAINED AT ALL.                     )

: GET-TO  CR ." END OF DATA=" INPUT
   0= IF RAM-END ! ELSE ERROR-EXIT THEN ;

: GET-TARGET  CR ." TARGET ADDRESS=" INPUT
   0= IF EPROM-START ! ELSE ERROR-EXIT THEN ;

: GET-TYPE     ( THIS WORD GETS THE EPROM TYPE NUMBER)
               ( FROM THE OPERATOR AND CHECKS TO MAKE)
               ( SURE THAT THE NUMBER IT RECEIVES    )
               ( AGREES WITH THE SETTING OF THE      )
               ( SWITCH.  IF NOT, IT WILL CONTINUE TO)
               ( ASK FOR THE EPROM TYPE UNTIL THE    )
               ( NUMBER IT RECEIVES AGREES WITH THE  )
               ( SETTING OF THE SWITCH.  THIS IS JUST)
               ( ONE MORE WAY TO IDIOT PROOF A SYSTEM)

   1 BEGIN
    CR ." EPROM TYPE=" INPUT
    DROP DUP 2716 =
    IF
       0 EPROM-TYPE ! DROP
    ELSE
       DUP 2532 =
     IF
         1 EPROM-TYPE ! DROP
     ELSE DROP
          DROP 0
     THEN
    THEN
    IF
       CHECK-TYPE
       EPROM-TYPE @ =
    ELSE
     0
    THEN
   UNTIL ;
```

FREQUENCY COUNTER

Randy Dumse
Michael Dalessio
Rockwell International

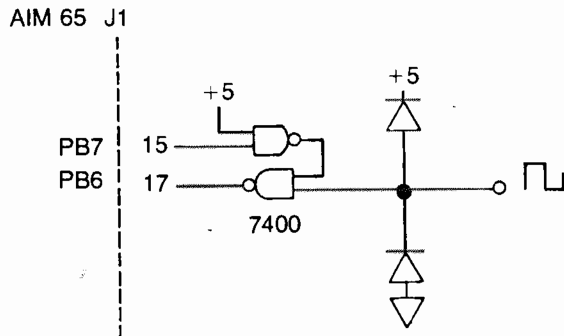
This program uses the unique features of the R6522 timers on the AIM 65 microcomputer and the power of the FORTH language to convert the AIM 65 into a frequency counter. Accuracy is good to the microsecond, the highest frequency readable being 500 KHz. Very minimal hardware is required to complete the system. Two versions are shown. The first uses two signal diodes to clamp the swing of the input in TTL ranges (the input must be >2.5V Peak to Peak). No special gating is provided so the counting window is switchable only for lower frequencies. The second circuit also uses the clamping diodes but has additional components to shape the counting window to exactly 50,000 μs (1/20th of a second) and should give accurate readings of frequency on signals to 500,000 cycles per second with a resolution of 20 cycles per second.



diodes can be most any germanium signal types such as 1N34.

more ⟶

```
: CHECK-ADDRESS      ( THIS ROUTINE CHECKS THE      )
                     ( ADDRESS RANGE TO MAKE SURE )
                     ( THAT IS CONSISTENT WITH THE)
                     ( EPROM TYPE AND CONSISTS OF )
                     ( AT LEAST ONE DATA BYTE      )

     RAM-END @ RAM-START @ -
     EPROM-START @ + DUP
     1 <
     IF
        DROP ERROR-EXIT
     THEN
        EPROM-TYPE?
     IF
        1000 SWAP -
     ELSE
        800 SWAP -
     THEN
     1 <
     IF
        ERROR-EXIT
     THEN ;

: PROGRAM     ( THIS IS THE MAIN ENTRY POINT FOR    )
              ( THE EPROM PROGRAMMER SOFTWARE.      )
              ( IT IS SELF-PROMPTING AND            )
              ( REASONABLY SELF EXPLANATORY.        )

     HEX H-ON GET-FROM GET-TO GET-TARGET
     SETUP-PORTB GET-TYPE CR H-OFF
     ADJ-EPROM-START CHECK-ADDRESSES
     CHECK-CLEAR BURN-IT VERIFY-CORRECT ;

FINIS
```



The FORTH program uses seven major word definitions to accomplish the task. The word RUN performs initialization (INIT) and then begins a loop (BEGIN) consisting of counting the number of cycles occurring in a 1/20th of a second interval (GETVAL), then multiplying by 20 to give cycles per second, and displaying the result (FREQ). The keyboard is checked for a key down (? TERMINAL) and the loop continued if no key is down (UNTIL). The word INIT outputs a carriage return (CR) to clear the display and sets the modes on Timer 1 and Timer 2 in the Auxiliary Control Register (ACR) of the R6522. The word GETVAL starts Timer 2 counting cycles and Timer 1 counting 1/20th of a second (START-TIMERS), waits for Timer 1 to time out (WAIT-ON-T1), and then reads the counts from Timer 2 (READ-COUNTS). The word FREQ takes the value from the stack, multiplies it by 20 and prints it with a label. The word START-TIMERS simply puts on $FFFF in Timer 2 (−1) and a count of 50,000 in Timer 1 which begins counting. The word WAIT-ON-T1 polls the Interrupt Flag Register (IFR) until the T1 bit is set. The word READ-COUNTS places an $FFFF on the stack; reads timer 2 and subtracts that value from $FFFF. The difference is the number of cycles counted on the input during the 1/20th of a second period.

As an exercise left to the reader, two improvements would be very desirable to make. The first would be to replace the word CR in the definition of FREQ with a code segment that calls the subroutine CLR ($EB44) in the AIM 65 monitor. This would keep the frequency from being printed each time taken. The second would be to write new definitions that would automatically scale the amount of time spent counting the input pulses until a given number of significant digits were assured.

```
( FREQUENCY COUNTER PROGRAM BY R. DUMSE)
DECIMAL

50000 CONSTANT 1/20TH-SEC

HEX
A004 CONSTANT TIMER1
A008 CONSTANT TIMER2
A00D CONSTANT IFR
A00B CONSTANT ACR

: INIT   ( SET TIMER CONTROLS IN ACR)
      CR A0 ACR C! ;

: START-TIMERS   ( PUT $FFFF INTO TIMER2)
      -1 TIMER2 !
      1/20TH-SEC TIMER1 ! ; ( SET TIMER1 FOR 50000 MS)

: WAIT-ON-T1
     BEGIN
      IFR C@
      40 AND
     UNTIL ;

: READ-COUNTS   ( $FFFF-TIMER2= COUNTS LEFT ON STACK)
     -1 TIMER2 @ - ;

: GETVAL
     START-TIMERS WAIT-ON-T1 READ-COUNTS ;
```

more→

RANDOM NUMBER GENERATOR

RNG's come in handy for a number of applications from games to systems modelling. Here is one which uses the countdown timer in the R6522 to obtain the "seed" value.

```
HEX
Ø VARIABLE SEED
: SETSEED
    A004 @ SEED ! ;
: RND
    SEED @ 4000 AND 0=
    SEED @ 2 * DUP >R
    4000 AND 0= XOR R>
    OR DUP SEED ! ;
: RANDOM
    RND 7FFF AND SWAP MOD;
```

To define these functions, make sure you're in HEX number mode.

To use this function, execute SETSEED in the beginning of your program to initialize the SEED variable.

From then on, whenever you need a positive random number, execute RANDOM with a range value on the stack. The random number will be returned on the stack.

If you need a random number between 0 and 6, place 7 on the stack and execute RANDOM.

```
7 RANDOM
```

If you now examine the stack, you'll find a number between 0 and 6.

If you can't use 0, simply add one to the result.

```
7 RANDOM 1+
```

Now the random number will be in a range from 1 through 7.

SYSTEM SPOTLIGHT

Randomatic Data Systems Inc. recently adapted an AIM 65 and wrote a program to computerize their product, a card filing and retrieval system. What the system does, and how AIM 65 now helps do it, might be of interest to many INTERACTIVE readers.

To begin with, Randomatic has for years made and marketed a system which provides a means of automatically retrieving randomly filed cards very quickly. The filing system accommodates cards of all sizes and materials—microfilm aperture cards, ledger cards, microfiche, etc. Each card to be filed is coded with edge notches.

Retrieval is accomplished by entering a code into the system. This causes bars in the selector trays to be raised. The bars match the edge coding, and, the selected cards are ejected so they "pop up" above the rest of the cards. They're then removed manually.

Cards can be filed and re-filed singly or in groups, at random, or under any sort of system you want. Up to 1,500 cards can be stored in a selector tray and up to ten trays may be searched simultaneously.

The Randomatic system has several obvious advantages. It provides instant retrieval. It eliminates misfiling, since selection is by code and requires no sequential filing. It can eliminate duplicate files by allowing cards to be retrieved by more than one criteria.

The success of the system depends upon accurate coding of the cards. The Randomatic system uses six fields of coding. With up to ten variations (0 through 9) in each of the six fields, 999,999 possible combinations are provided.

The system allows for thirty notch positions along any card edge. Each of the six fields has five possible notch positions. Using a 2 out of 5 code system, these thirty notch positions provide ten selection variables in each of six fields.

Until the AIM 65 was recently adapted, notching the edge of the cards and informing the system of the desired code to be retrieved was accomplished through a 10-button keyboard.

more→

```
DECIMAL
: FREQ   ( MULTIPLY STACK VALUE BY 20, LABEL AND PRINT)
     20 M* ." FREQ=" D. CR ;

: RUN
     INIT
      BEGIN
       GETVAL FREQ ?TERMINAL
      UNTIL ;

FINIS
```

In actual operation, a blank card is inserted into a punch unit, the desired six digit code is entered on the keyboard, the punch is activated and the card ends up notched and ready for filing. For retrieval, the code is entered in the keyboard, an "operate" key is depressed, and the system "pops up" the proper card(s) automatically.

The changeover to the AIM 65 has been very straightforward. The AIM 65 replaces the 10-button keyboard and an interface unit has been added between the AIM 65 and the system. All other parts of the system remain the same.

The AIM 65 used by Randomatic has 1K of RAM with two programmed 2716 EPROMs replacing the two R2332 monitor ROMs. It uses a Condor AA524 power supply and an enclosure from the Enclosure Group. The interface and the programming are from Randomatic.

The new computerized system will accept a code of up to 20 alphanumeric characters instead of only the six-digit one. It allows the operator to use whatever fits his normal system—names, word descriptions, part numbers, etc.

Since the Randomatic system is still searching for only a six digit code, a mathematical formula programmed into the AIM 65 converts the 20-character input into a six digit number. The formula makes sure that the same input always results in the same output. There is the possibility that the converted numbers may be redundant but this doesn't seriously affect the Randomatic system results.

Use of the AIM 65 offers other benefits compared to the old 10-key board. A display is now provided so the operator can check and verify a code before activating the system. And, the AIM 65 printer can be used to keep a record of inputs and outputs.

The AIM 65 also "thinks" a little when used with the Randomatic interface unit. As an example, the operator can pre-determine one or more selector trays, if desired. In this example, if the code number was CR 765432 FS 2 , the interface unit would first accept the file select (FS) number, turn to tray 2, then transmit the CR number of 765432 to select a card.

So, this shows how one company has adapted an AIM 65 to control an electromechanical system. It's a unique application and one that might be of interest to INTERACTIVE readers, both in its own right and as a stimulus to think of other AIM 65 applications.

Our purpose isn't to sell Randomatic products. Rockwell International has no connection or interest in Randomatic. However, Randomatic does have products for sale that might be of interest.

A customized AIM 65 can be purchased as a "stand alone" to convert any 20-character alphanumeric input into a six digit output, displaying the codes as required and holding them in memory. Or, the program for accomplishing this can be acquired from Randomatic. And, of course, a complete Randomatic filing and retrieval system is also for sale.

Randomatic desk console, with two recessed selector trays to the left and the AIM 65 controller to the right. The slot in front of the AIM 65 is the punch unit. The power supply and interface are inside the console.

L.G. Stine, president of Randomatic, has also expressed an interest in providing an interface mechanism, on a custom basis, between AIM 65 and almost any electromechanical equipment.

For more information on Randomatic products or services, please contact the company direct:

Randomatic Data Systems Inc.
216 Robbins Ave.
Trenton, New Jersey 08638
(609) 833-4860.

And, if you have an interesting AIM 65 application you would like to share with others, please send it to:

Editor, Interactive RC55
Rockwell International
P.O. Box 3669
Anaheim, CA 92803
(714) 632-3729

USING SMUDGE

Get out your AIM 65 FORTH User Manual and look at page 5-17. Notice the smudge flag in the second illustration? The use of that flag has presented me with some difficulty so you may be having trouble with it also.

Basically, it's used to prevent the dictionary search routine from finding the name of the word that's presently being compiled. This would only be a problem if you were attempting to redefine another routine and perhaps change the function slightly.

For example, if you wanted to change the word MON to print out a message before leaving FORTH, you might say:

```
: MON . " BYE BYE FORTH " MON ;
```

In FIG-FORTH (AIM-65 FORTH is based on FIG-FORTH) if it weren't for the smudge flag, the dictionary search would find the name of the routine you were defining and think you meant to perform a recursion (when a routine calls itself) instead of referencing the MON word already built into the FORTH Rom.

So, when a routine is starting to be defined, the smudge flag gets set automatically to prevent the new name from showing up in the search. The flag is then cleared when the new function compiles successfully.

Ahhh . . . but what happens if the new function DOESN'T compile successfully. Well, I'm glad you asked that question because that's precisely the reason I wrote this piece.

Try compiling this definition.

```
: TEST1 GHBFM ;
```

Unless you had already defined a function called GHBFM, the system should choke and wonder what the heck you're trying to do. But, since FORTH is very well-mannered it will only say—

```
GHBFM ?
```

Now, try to forget TEST1.

```
FORGET TEST1 <return>
```

Since the smudge flag got set at the beginning of the definition and the definition didn't compile correctly, that flag never got reset. The word TEST1 will show up in a VLIST—try it—but can't be forgotten.

Fortunately, FORTH provides us with an antidote for this seemingly strange and frustrating behavior.

Type in—

```
SMUDGE <return>
```

After FORTH indicates that it's now OK, try to FORGET TEST1 again.

Success!!!

Now, that wasn't so difficult, was it? By the way—SMUDGE only works on the last word defined in the dictionary.

NUMBER INPUT FUNCTION

If you're working on a FORTH program that needs to prompt the user for some number input, it would be handy to have a generalized number input routine.

There is a routine called INPUT (appropriately enough) which is defined on page 5-15 of the AIM-65 FORTH User Manual, but it needs some minor changes for it to suit our purposes.

As it's defined, INPUT returns the number typed in as a double precision value and the address of the location just beyond where the ASCII string of that number (before it was converted to a number) is located in memory.

A slight mod will eliminate the address and make the routine a bit more useful.

Here is the modified form of INPUT:

```
: INPUT
    PAD 10 EXPECT
    0 0 PAD 1-
    (NUMBER) DROP ;
```

Since INPUT returns a double precision number, we could DROP the top number on return from INPUT to leave a single-precision number, or we could test it for zero to make sure the user didn't type in a number that was too large. We can easily do this test by using the Ø= function after the call to INPUT.

```
: TEST
    CR . " INPUT NUMBER " INPUT
    0= NOT IF CR . " TOO LARGE "
    DROP THEN ;
```

: TEST

CR . " INPUT NUMBER "	output carriage return and message
INPUT	get input from keyboard
0=	if top stack value equals zero, then number is single precision
NOT	invert truth value
IF	if truth value flag = 1 then number was too large
CR . " TOO LARGE "	output carriage return and error message
DROP	drop rest of number
THEN ;	if truth value flag = Ø, branch to THEN and exit with single precision number on stack

See how easy it is to add new functions to FORTH?

DUMPING BASIC

A. Ward
192 Greenock Road,
Largs,
Ayrshire,
Scotland, U.K.

It is sometimes convenient to record BASIC on cassette tape as if it were a program in MACHINE CODE which of course it is in a sense. All that is necessary is to record the zero page usage as well as the program itself which starts near the beginning of page 2 of memory ($0212). However, *do not* record page 1 of memory as trying to read this back can cause problems and page 1 is not required.

An example may help to make the position clear. Let's imagine you originally answered the BASIC prompt "Memory Size?" with 1024, this means of course that your memory up to the top of page 3 ($03FF) has been allocated to BASIC. When you want to record the program (as BASIC !) treat the whole operation as a machine code dump (under a suitable file name such as BFILE) by entering Monitor and using the "D" key. First dump page 0 ($000–$00FF) and then reply to the prompt "More?" with yes "Y" and proceed to dump pages 2, 3 ($0200–$03FF). If you have any machine code subroutines associated with Basic you can dump these also by continuing to answer "Y" and giving their address range.

When reloading your program enter Monitor then use the "L" key and ask for BFILE. To enter your BASIC program hit the "6" key for "warm start" and there you are. The advantages of this procedure are that you don't have to use the remote control on the cassette recorder and a mixed program of Basic and Machine Code subroutines can all be under the one file name.

CASSETTE PROBLEM

I have received and confirmed reports from two separate sources that there is a bug somewhere in the cassette interface software in the AIM 65 system.

There are at least three file lengths that cause the system to "hang-up" when reading.

If you dump a file with a length of hex 161, 162 or 163 bytes, the data will dump and load o.k. But, the program never returns from the LOAD sequence. The data does GET LOADED but the program hangs up.

The problem has something to do with the number of file characters that are used to "pad" the last block to make it 80 bytes long. If the number of pad characters ($00) are less than four, the program will never exit the routine of $E321 (LOAD 4) in the monitor.

There may be other lengths (besides $161, $162 and $163) that cause this problem, but I haven't been able to find them. BEWARE!!!

COMPAS, WHERE ARE YOU?

I have received several phone calls from folks who claim that COMPAS Microsystem (supplier for AIM 65 accessories) is now defunct.

To verify this, I tried to call their number (got a strange busy signal) and then attempted to get their number from the local (Ames, Iowa) information operator. She had no listing for any such company.

So, if you were considering using their products, I'm sure glad you read this.

NEWSLETTER EDITOR
ROCKWELL INTERNATIONAL
P.O. Box 3669, RC55
Anaheim, CA 92803 U.S.A.

Bulk Rate
U.S. POSTAGE
RATE
Santa Ana Calif.
PERMIT NO. 15