AUGUST/SEPTEMBER, 1980. ISSUE 3. $2.00

# compute II.

**The Single-Board COMPUTE™**

## Table of Contents



## Advertisers Index



**Staff of compute II.:**
Robert C. Lock, Editor/Publisher
Carol Holmquist Lock, Circulation Manager.
Larry Isaacs, Software/Hardware Lab.
Joretta Klepfer, Editorial Assistant.

compute II. receives continuing assistance from the following persons:
Harvey B. Herman, University of North Carolina at Greensboro, Dept. of Chemistry. Editorial Assistance.
Gary Dean, The Design Group, Greensboro, N.C., Art Direction/Design Consultation.
Jim Butterfield, Toronto. Editorial Assistance.

The following writers contribute on a regular basis as columnists:
Eric Rehnke, 1067 Jadestone Lane, Corona, CA, 97120. Associate Editor
Dann McCreary, Box 16435-Y, San Diego, CA, 92116.
Gene Zumchak, 1700 Niagara Street, Buffalo, NY, 14207

**Application to mail at controlled circulation postage rate is pending at Greensboro, North Carolina. Postmaster: Send change of address to compute II, Post Office Box 5406, Greensboro, NC 27403.**
compute II. is published by Small System Services, Inc., 900-902 Spring Garden Street, Greensboro, North Carolina 27403. Telephone: (919) 272-4867. compute II. is published six times each year on a bimonthly schedule. Subscription cost for one year is $9.00. compute II. is available by subscription or through retail dealer sales. Subscription prices higher outside the US. (See below)

Address all manuscripts and correspondence to compute II. Post Office Box 5406, Greensboro, N.C. 27403. Materials (advertising art work, hardware, etc.) should be addressed to compute II., 900 Spring Garden Street, Greensboro, N.C. 27403.


Authors of manuscripts warrant that all materials submitted to compute II. are original materials with full ownership rights resident in said authors. By submitting articles to compute II. authors acknowledge that such materials, upon acceptance for publication, become the exclusive property of Small System Services, Inc. Unsolicited manuscripts not accepted for publication by compute II. will be returned if author provides a self-addressed, stamped envelope. Program listings should be provided in printed form as well as machine readable form. Articles should be furnished as typed copy with double spacing. Each page of your article should bear the title of the article, date and name of the author.

**Address all articles, circulation questions and other inquiries to:**
**compute II.**
**P.O. Box 5406**
**Greensboro, NC 27403**
**(919) 272-4867**

# The Editor's Notes

Robert Lock
Publisher/Editor

# IMPORTANT ANNOUNCEMENT!

**COMPUTE** and **compute II** are merging into one, high quality, monthly magazine.

This is the last issue of **compute II.** Your next magazine, **COMPUTE!,** will arrive in November.

## The Timetable

The merger is effective with the November/December issue of **COMPUTE!** In January, we're monthly! Each month you'll receive the same quality of single board information we've been providing in **compute II.**

## Don't Despair!

New **COMPUTE!** will contain a healthy Single-Board Computer Gazette, continuing to provide you with useful, up-to-date information. We're committed to continuing to provide as much or more information in two monthly issues as we've been providing in one bimonthly issue.

## OSI Moves Out

New **COMPUTE!** will have an OSI Gazette for you OSI owners, with the Single-Board Computer Gazette devoting its space to KIM, SYM and AIM.

## And What About the 1802?

Dann McCreary will continue his column, emphasizing the areas of communication between the 6502 and the 1802.

That's the new **COMPUTE!** Your subscription will be adjusted to make sure you get your six issues of **computeII/COMPUTE!** We'll explain the process in Issue 7 of **COMPUTE!** (November/December: the copy you'll receive next time.) ©

# The Single-Board 6502

Eric Rehnke

Well, I finally did it. Got myself an APPLE II to play with. No, I'm not abandoning KIM. Just wanted to see what all the hullaballoo was about.

Sure is easier to demonstrate than KIM. Who wants to see an assembler when they can see some neat video-arcade like games in action?

### ...And Ever The Twain Shall Meet

While I was preparing to take an AIM 65 system to the local computer club for a demo, it became painfully obvious that I would either have to build a special version of the sound generator board for my AIM 65/MTU system or, somehow, adapt the sound board that was built for my KIM-1/HDE system. I decided to adapt rather than fight with a new board from the ground up. Luckily, in my MTU card cage, the bottom row of slots are not used because of the way the MTU backplane board has been raised to accomodate the AIM 65. Also, the spacing between the card guide on the right hand side of the cage and the edge connector turned out to be just perfect for supporting the 4.5''x6.0'' HDE size card. It was almost like the cage was designed to accomodate the standard 4.5''x6.0'' size prototyping cards. (Keep this in mind when you need a quick and cheap proto board in your MTU system as they can be obtained for less than $10).

Interestingly, a week later, I needed to adapt an MTU card (the Visible Memory graphics board) to my KIM-1/HDE system. Because of the size of the Visible Memory card it had to be mounted outside the HDE cage. A cheap 4.5''x6.0'' proto board was installed in the HDE system with ribbon cable to extend the bus out to a 44-pin card edge connector which plugged on to the MTU card.

Both transplants are doing fine, thank you. And, on nights off from doing serious development work, big KIM can relax with some pleasing graphics as well as some interesting sound effects.

Here's how to cross pollinate your own system:

| KIM-4/HDE BACKPLANE | SIGNAL NAME | MTU/AIM-65 BACKPLANE |
|---|---|---|
| B | AB0 | A |
| C | AB1 | B |
| D | AB2 | C |
| E | AB3 | D |
| F | AB4 | E |
| H | AB5 | F |
| J | AB6 | H |
| K | AB7 | J |
| L | AB8 | K |
| M | AB9 | L |
| N | AB10 | M |
| P | AB11 | N |
| R | AB12 | P |
| S | AB13 | R |
| T | AB14 | S |
| U | AB15 | T |
| 8 | DB7 | 8 |
| 9 | DB6 | 9 |
| 10 | DB5 | 10 |
| 11 | DB4 | 11 |
| 12 | DB3 | 12 |
| 13 | DB2 | 13 |
| 14 | DB1 | 14 |
| 15 | DB0 | 15 |
| 19 | +8 | 18 |
| 22 | GND | 22 |
| 7 | RES | 7 |
| X | 02 | Y |
| 17 | +16 | X |
| W | R/W | V |

**Signal Conversion Table**

### Sound Chip Driver

As promised, here is the low level driver software for interfacing a 6522 VIA to the General Instruments AY3-8910 Programmable Sound Generator (PSG). The explosion routine is included to satisfy yourself that the interface works correctly. The clock circuit was duplicated from page 33 of the PSG manual.

The 1 MHZ system clock (01 or 02) could have been used to save a few dollars but then all example values given in the documentation would have to be recalculated. Building the suggested clock input circuitry seemed to be the easier of the two alternatives since I had the parts on hand anyway.

The audio output circuitry was duplicated from page 6 of the PSG manual and used to feed one of the cheap (under $10) Radio Shack speaker/amplifiers.

Connections from the 6522 to the PSG are similar to the scheme presented on page 43 of the PSG manual except that BC2 (pin 28) is connected to +5 volts (not PB1), and BDIR (pin 27) is connected to PB1 (not PB2).

This way, three additional PSG chips can be connected to the 6522 as my drawing indicated in issue #3 of COMPUTE (page 104).

I should mention that there was one thing about the PSG manual that really messed me up for awhile. All the register numbers and values are expressed in octal! Once I realized this, programming the chip went much easier.

OK. I've shown you how to hook up this neat chip and even threw in some software to get you going.

What kinds of interesting sounds can you come up with? Can you program wind chimes or bells? I'll publish any neat sound programs.

```
2000                ;SOUND CHIP DRIVER PROGRAM
2000                ;WRITTEN BY ERIC C. REHNKE
2000
2000                ;6522 DEFINITIONS
2000
2000                IOBASE =                $0810
2000
2000                ORB     =IOBASE
2000                DDRB    =IOBASE+2
2000                DDRA    =IOBASE+3
2000                OREGA   =IOBASE+15
2000
2000                PRTBYT =$1E3B           ;KIM HEX TO ASCII ROUTINE
2000                CRLF    =$1E2F
2000                OUTSP   =$1E9E
2000
2000                STBUF   =$2500
2000                ;MAINLINE ROUTINE
2000                        *=$2000
2000                ;
2000                ;******OUTPUT TO THE SOUND CHIP******
2000                ;IN ORDER TO SET A SOUND CHIP REGISTER TO
2000                ;A PARTICULAR VALUE, ENTER THIS ROUTINE WITH
2000                ;THE 'X' REGISTER CONTAINING THE SOUND CHIP
2000                ;REGISTER NUMBER AND THE ACCUMULATOR CONTAINING
2000                ;THE DATA TO BE LOADED INTO THAT REGISTER.
2000                ;
2000  A8            OUTPUT TAY              ;SAVE DATA
2001  20 10 20              JSR LATCH
2004  20 26 20              JSR WRITE
2007  60                    RTS
2008                ;
2008                ;******INPUT FROM THE SOUND CHIP******
2008                ;IN ORDER TO READ THE CONTENTS OF A
2008                ;PARTICULAR SOUND CHIP REGISTER, ENTER
2008                ;THIS ROUTINE WITH THE SOUND CHIP REGISTER
2008                ;NUMBER IN THE 'X' REGISTER. UPON RETURN,
2008                ;THE DESIRED REGISTER DATA WILL BE FOUND
2008                ;IN THE ACCUMULATOR
2008                ;
2008  20 10 20      INPUT   JSR LATCH
200B  20 35 20              JSR READ
200E  98                    TYA             ;RESTORE DATA
200F  60                    RTS
2010
2010                ;THE 'LATCH' ROUTINE SIMPLY LATCHES
2010                ;THE SOUND CHIP REGISTER NUMBER INTO
2010                ;THE SOUND CHIP ADDRESS REGISTER FOR
2010                ;A SUBSEQUENT READ OR WRITE.
2010                ;
2010  A9 FF         LATCH   LDA #$FF        ;MAKE IT ALL OUTPUTS
2012  8D 13 08              STA DDRA
2015  8D 12 08              STA DDRB
```

HDE inc.

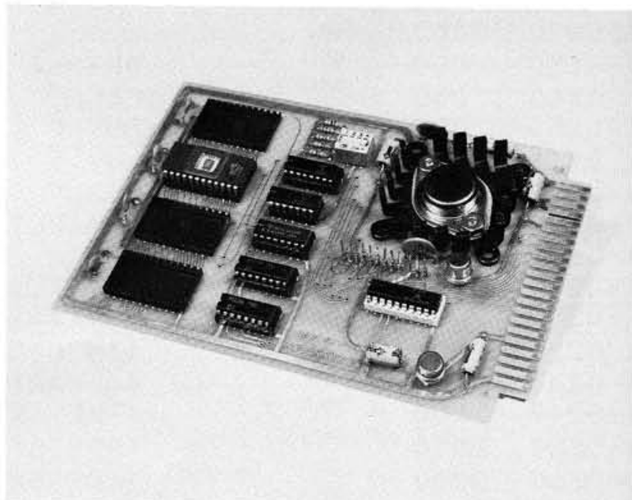
BOX 120
ALLAMUCHY, N.J. 07820
201-362-6574

HUDSON DIGITAL ELECTRONICS INC.

# ANNOUNCING THREE NEW PRODUCTS

## 1. 4/8K EPROM CARD

A 4/8K EPROM card, featuring on-board jumper selection of 2708 or 2716 EPROMS. Compact, industry standard 4½ x 6½" card size with on board regulation of all required voltages. Uses the KIM-4 standard 44-pin bus. (EPROMS not included)
HDE DM 816-P8 — $165.00



DM 816-P8 — (EPROMS not included)

## 2. HDE DISK BASIC (Now it's KIM's turn)

HDE Disk Basic has been designed so that the 6502 Basic versions for SYM, KIM, TIM and AIM are subsets, thereby allowing program transfers without any modification in most instances. For program development we've included and enhanced the editing features available in our text editor, TED. Other facilities include: ON ERROR GOTO ...; IF ... THEN ... ELSE; LINE INPUT; PRINT USING; AUTO line numbering; renumbering; hex value input and much more. Disk capabilities include: SAVE, LOAD, RUN, LINK, CHAIN and sequential and random input/output. KIM version available now for HDE Disk based systems — $175.00

## 3. HDE 'AID' – An Advanced Interactive Disassembler

The Advanced Interactive Disassembler, designed by PCS, Progressive Computer Software, is a resident, two pass disassembler that creates TED compatible source files, with labels assigned to all address references. Addresses external to the object file limits are defined as equates in the source.

### AID Builds Source Files for All Your Object Programs

The creation of source files of any object program is limited only by the size of the object program, the symbol table and the user defined source file buffer. AID will save interim source files to disk. KIM, TIM, SYM and AIM HDE Disk Based versions – $95.00

```
2018  8E 1F 08            STX OREGA
201B  A9 03                LDA #3          ;STROBE IN THE REG ADDRESS
201D  8D 10 08             STA ORB
2020  A9 00                LDA #0
2022  8D 10 08             STA ORB
2025  60                   RTS
2026
2026              ;THE 'WRITE' ROUTINE ASSUMES THE
2026              ;PROPER REGISTER VALUE HAS ALREADY
2026              ;SETUP IN THE SOUND CHIP AND LOADS
2026              ;THE PROPER SOUND CHIP REGISTER WITH
2026              ;THE CONTENTS OF THE ACCUMULATOR.
2026              ;
2026  98          WRITE    TYA
2027  8D 1F 08             STA OREGA
202A  A9 02                LDA #2
202C  8D 10 08             STA ORB
202F  A9 00                LDA #0
2031  8D 10 08             STA ORB
2034  60                   RTS
2035
2035              ;THE 'READ' ROUTINE ASSUMES THE PROPER
2035              ;SOUND REGISTER CHIP HAS BEEN SELECTED
2035              ;AND READS THAT REGISTER INTO THE
2035              ;ACCUMULATOR.
2035              ;
2035  A9 00       READ     LDA #0
2037  8D 13 08             STA DDRA
203A  A9 01                LDA #1
203C  8D 10 08             STA ORB
203F  AD 1F 08             LDA OREGA       ;GET DATA
2042  A8                   TAY
2043  A9 00                LDA #0
2045  8D 10 08             STA ORB
2048  60                   RTS
2049
2049              ;THE 'CLEAR' ROUTINE ZEROS ALL THE REGISTERS
2049              ;IN THE SOUND CHIP.
2049              ;
2049  20 73 20    CLEAR    JSR INITS
204C  A2 00                LDX #0
204E  A9 00       DOIT     LDA #0
2050  20 00 20             JSR OUTPUT
2053  E8                   INX
2054  E0 11                CPX #17
2056  D0 F6                BNE DOIT
2058  00                   BRK
2059
2059              ;THE 'CHECK' DUMPS THE CONTENTS OF
2059              ;ALL THE SOUND CHIP REGISTERS TO
2059              ;THE SERIAL TERMINAL.
2059              ;
2059  20 73 20    CHECK    JSR INITS
205C  20 2F 1E             JSR CRLF
205F  A2 00                LDX #0
2061  20 08 20    GETIT    JSR INPUT
2064  20 3B 1E             JSR PRTBYT
2067  20 9E 1E             JSR OUTSP
206A  E8                   INX
```

```
01-1000  206B  E0 11              CPX #17
01-1010  206D  D0 F2              BNE GETIT
01-1020  206F  20 2F 1E           JSR CRLF
01-1030  2072  00                 BRK
01-1040  2073
01-1060  2073
01-1061  2073              ;THE 'INITS' ROUTINE SETS UP THE
01-1062  2073              ;6522 WITH PB0-PB7 AS OUTPUTS
01-1063  2073              ;AND WRITES A $00 TO THAT PORT.
01-1064  2073              ;
01-1070  2073  A9 FF       INITS  LDA #$FF
01-1080  2075  8D 12 08           STA DDRB
01-1090  2078  A9 00              LDA #0
01-1100  207A  8D 10 08           STA ORB
01-1110  207D  60                 RTS
01-1120  207E
01-1130  207E
01-1140  207E
01-1150  207E              ;EXPLOSION SOUND EFFECT
01-1160  207E
01-1170  207E  A9 00       EXPLOS LDA #$0
01-1180  2080  A2 06              LDX #6           ;SETUP REG 6
01-1190  2082  20 00 20           JSR OUTPUT
01-1200  2085  A9 07              LDA #$7
01-1210  2087  A2 07              LDX #7           ;SAME FOR REG 7
01-1220  2089  20 00 20           JSR OUTPUT
01-1230  208C  A9 10              LDA #$10
01-1240  208E  A2 08              LDX #8
01-1250  2090  20 00 20           JSR OUTPUT
01-1260  2093  A9 38              LDA #$38
01-1270  2095  A2 0C              LDX #12
01-1280  2097  20 00 20           JSR OUTPUT
01-1290  209A  A9 10              LDA #$10
01-1300  209C  A2 09              LDX #9
01-1310  209E  20 00 20           JSR OUTPUT
01-1320  20A1  A9 10              LDA #$10
01-1330  20A3  A2 0A              LDX #10
01-1340  20A5  20 00 20           JSR OUTPUT
01-1350  20A8  A9 00              LDA #0
01-1360  20AA  A2 0D              LDX #13
01-1370  20AC  20 00 20           JSR OUTPUT
01-1380  20AF  00                 BRK
01-1390  20B0
01-1400  20B0
01-1410  20B0              ;THIS SECTION LOADS THE SOUND CHIP
01-1420  20B0              ;WITH THE FIRST 16 BYTES STARTING AT
01-1430  20B0              ;LOCATION $2500
01-1440  20B0
01-1442  20B0
01-1450  20B0  A2 00       LOAD   LDX #0
01-1460  20B2  BD 00 25    LOOP1  LDA STBUF,X  ;NOW GET DATA
01-1470  20B5  20 00 20           JSR OUTPUT
01-1480  20B8  E8                 INX
01-1490  20B9  E0 11              CPX #17      ;DONE YET?
01-1500  20BB  D0 F5              BNE LOOP1
01-1510  20BD  4C 59 20           JMP CHECK    ;DUMP THE CONTENTS
01-1520  20C0                                        OF THE CHIP
01-1530  20C0                     .END
```

**More In Store**

Now that we have sound output, it's only logical that we should have some sort of analog input. Besides, if we only hook one sound chip to the 6522 we have plenty of lines left--so let's use 'em. I happen to have a NATIONAL ADC0816 laying around that's just waiting to do something.

It's an 8 bit A/D converter with 16 analog inputs. The conversion time is around 100 us and it runs on a single 5 volt supply. Ideal for joysticks and other analog devices.

Look for it in an upcoming column.

**You Got Time?**

What about the date? If your micro has need for the time and date, you'll be glad to hear that a new 18 pin, CMOS clock/calendar chip (MSM 5832) has been introduced by OKI Semiconductor (1333 Lawrence Expressway, Santa Clara, CA 95051 (408-984-4842)) that can be easily interfaced to a 6522 VIA. In fact, it was made to interface with micros.

The MSM 5832 chip and necessary crystal (32.768 KHZ) cost under $15 and is now generally available. If you can't find it locally, I got mine at Concord Computer Components (1973 So. State College, Anaheim, CA 92806 (714) 937-0637).

**More On Communications**

If you're interested in computer communications, two magazines recently had articles which will feed your enthusiasm.

Byte magazine (June 1980) had two useful articles which you will want to read.

The first article (on page 24) showed how to build a complete modem with pre-aligned filter modules which eliminates the need for complicated adjustments. The 6860 modem chip was used which is a perfect match for the new 6551 ACIA chip which is being manufactured by Rockwell and Synertek.

Page 140 (of the same issue) presents two methods of having KIM dial your phone. The first method uses the conventional relay approach while the second one uses a D/A converter (just like the one on the Micro Technology Unlimited D/A board) to generate and mix the two signals necessary to create the touch-tone pair.

Doctor Dobbs Journal (June/July 1980) devoted part of an issue to the subject of networking which included an update on the PCNET efforts of Dave Caulkins, several articles on networking and a description of MCALL-C, another communications protocol.

They also had a directory of phone numbers for 144 computerized bulletin board systems.

Lots of things are happening in this area of personal computing and commercial computing, as well. If you're looking for a possible future career in some area of computing, telecommunications is a good choice.

**HDE Software Bank**

Hudson Digital Electronics (Box 120, Allamuchy, N.J. 07820 (201) 362-6574) has just concluded negotiations which would put Progressive Computer Software Inc. (405 Carbin Rd., York, PA 17403) in charge of maintaining the HDE Users Library.

The plan is to offer utility and applications programs available at a nominal disk copying charge.

Contact HDE and/or Progressive for more details.

**6502 High-Level Languages Available**

Several high level languages are available from the good folks at 6502 Program Exchange (2920 Moana, Reno, NV 89509). For AIM, KIM and SYM systems, they're offering FOCAL, TINY BASIC and XPLO (a compiler) as well as an editor and assembler.

These people have been around since the beginning and done much to help the 6502 attain its present popularity level.

Send $1 for their latest catalog. ©

**An Evaluation:** Marvin L. De Jong

# The FIRST MATE/SECOND MATE By Micromate

Ever since I began doing experimental work with 6502 single-board microcomputers, such as the KIM-1, SYM-1, and the AIM 65, I have looked for a neat and convenient way for my students and me to breadboard circuits to be interfaced to the microcomputer. The FIRST MATE/SECOND MATE combination by MicroMate, P.O. Box 50111, Indianapolis, IN 46256 will probably end my search. In my opinion, this system is an excellent way to prototype and interface circuits to the microcomputer. It will be of great interest to engineers, technicians and experiters as well to those of us involved in technical education.

The SECOND MATE is simply a 2½" by 3½" printed circuit board with a 22/44 pin edge connector on one side, and a set of 44 printed circuit pads that duplicate the application and expansion edges on the KIM-1, SYM-1 or AIM 65. Thus, the SECOND MATE is transparent to any other devices you may want to connect to your microcomputer. Finally, the SECOND MATE has a 40 pin connector that connects to a 40 pin DIP jumper that connects the SECOND MATE to the FIRST MATE by a 40-strand ribbon cable. The DIP jumper is about 6" long. Thus, the SECOND MATE is connected to the microcomputer with the usual 22/44 pin edge connector, and the FIRST MATE connects to 40 of the 44 lines that are available at these connectors.

The FIRST MATE is a 7½" square printed circuit board upon which is mounted an SK-10 breadboard, three 40-pin connectors, a position for a second SK-10 or another protoboard, four "universal" connectors for GND, +5V, +V and a -V supply. An LED indicates when power is applied, and several filter capacitors are also provided. The three 40-pin connectors on the FIRST MATE connect to either the expansion connector, the application connector or, if you have a SYM-1, the so-called AA connector. Suppose you wish to interface a circuit to the expansion connector on your microcomputer. The 40-pin DIP jumper is then connected to the 40-pin expansion connector on the FIRST MATE, while the SECOND MATE is plugged into the expansion port on the microcomputer. The eight data lines, the sixteen address lines, and eight control lines are then connected to labelled locations on the SK-10. Each labelled location allows up to five wires to be connected. The control lines are the usual ones, R/W, $0_2$, RES, NMI, IRQ, RDY, SYNC, and one device select line. (For the SYM-1, the device select is the 18 line. Some minor trace cutting and jumpering gives the CS8 line or another device select for the AIM 65.)

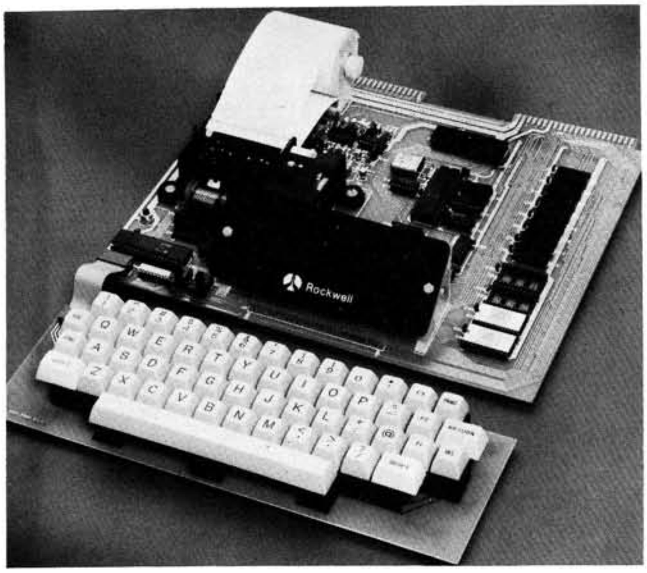
If the FIRST MATE/SECOND MATE are connected to the application port, then the eight pins of Port A and the seven pins of Port B may be accessed on the SK-10 at labelled positions. Note that both the SYM-1 and the KIM-1 do not allow a connection to PB6 at the applications port. If you want to use the FIRST MATE with an AIM 65 you will probably want to jumper PB6 to the SK-10 as well as the control lines CA1, CA2, CB1, and CB2 from the VIA. This would be quite simple, but it would eliminate (or duplicate) some pin functions for the AA connector on the SYM-1. Connections can also be made to the expansion port and the applications port simultaneously if two SECOND MATES and two ribbon cables are purchased.

The geometry of the First MATE was designed to mount on a SYM-1 with nylon spacers and screws. The FIRST MATE can probably be placed on a KIM-1 with no problems. For my AIM 65 I chose to build a little table consisting of two 4" X 8" pine legs and a 14" X 8" masonite perforated board for a top. This not only makes a dust cover for the AIM 65, but it also keeps me from yelling at the cat when he decides to sleep on my microcomputer. If the little table is made about 8" deep then the printer paper can be easily seen. The FIRST MATE can be bolted to the perf-board top. I think it made a neat system, allowing me to work directly over the microcomputer when I was breadboarding a circuit.

Clearly the FIRST MATE was designed for the SYM-1, but with a few simple modifications, some of which are suggested in the literature supplied with the FIRST MATE, it can be used with the AIM 65. No modifications are necessary for operation with the KIM-1. To put the FIRST MATE to the test, I breadboarded the simple stepper motor interface described in this issue. No modifications to the FIRST MATE were required for this circuit.

Although there are other breadboarding schemes available (see TERC, 575 Technology Sq., Cambridge, MA 02139 for other possibilities) that are not being evaluated here because I have little or no experience with them, I can wholeheartedly recommend that you examine the MicroMate system for $87.50. I think it is an excellent approach to circuit development. I would like to see an AIM 65 version of the system for sale, but the modifications are quite simple.

# AIM 65 BY ROCKWELL INTERNATIONAL



AIM 65 is fully assembled, tested and warranted. With the addition of a low cost, readily available power supply, it's ready to start working for you.

AIM 65 features on-board thermal printer and alphanumeric display, and a terminal-style keyboard. It has an addressing capability up to 65K bytes, and comes with a user-dedicated 1K or 4K RAM. Two installed 4K ROMS hold a powerful Advanced Interface Monitor program, and three spare sockets are included to expand on-board ROM or PROM up to 20K bytes.

An Application Connector provides for attaching a TTY and one or two audio cassette recorders, and gives external access to the user-dedicated general purpose I/O lines.

Also included as standard are a comprehensive AIM 65 User's Manual, a handy pocket reference card, an R6500 Hardware Manual, an R6500 Programming Manual and an AIM 65 schematic.

AIM 65 is packaged on two compact modules. The circuit module is 12 inches wide and 10 inches long, the keyboard module is 12 inches wide and 4 inches long. They are connected by a detachable cable.

**THERMAL PRINTER**
Most desired feature on low-cost microcomputer systems . . .
- Wide 20-column printout
- Versatile 5 x 7 dot matrix format
- Complete 64-character ASCII alphanumeric format
- Fast 120 lines per minute
- Quite thermal operation
- Proven reliability

**FULL-SIZE ALPHANUMERIC KEYBOARD**
Provides compatibility with system terminals . . .
- Standard 54 key, terminal-style layout
- 26 alphabetic characters
- 10 numeric characters
- 22 special characters
- 9 control functions
- 3 user-defined functions

**TRUE ALPHANUMERIC DISPLAY**
Provides legible and lengthy display . . .
- 20 characters wide
- 16-segment characters
- High contrast monolithic characters
- Complete 64-character ASCII alphanumeric format

**PROVEN R6500 MICROCOMPUTER SYSTEM DEVICES**
Reliable, high performance NMOS technology . . .
- R6502 Central Processing Unit (CPU), operating at 1 MHz. Has 65K address capability, 13 addressing modes and true index capability. Simple but powerful 56 instructions.
- Read/Write Memory, using R2114 Static RAM devices. Available in 1K byte and 4K byte versions.
- 8K Monitor Program Memory, using R2332 Static ROM devices. Has sockets to accept additional 2332 ROM or 2532 PROM devices, to expand on-board Program memory up to 20K bytes.
- R6532 RAM-Input/Output-Timer (RIOT) combination device. Multipurpose circuit for AIM 65 Monitor functions.
- Two R6522 Versatile Interface Adapter (VIA) devices, which support AIM 65 and user functions. Each VIA has two parallel and one serial 8-bit, bidirectional I/O ports, two 2-bit peripheral handshake control lines and two fully-programmable 16-bit interval timer/event counters.

**BUILT-IN EXPANSION CAPABILITY**
- 44-Pin Application Connector for peripheral add-ons
- 44-Pin Expansion Connector has full system bus
- Both connectors are KIM-1 compatible

**TTY AND AUDIO CASSETTE INTERFACES**
Standard interface to low-cost peripherals . . .
- 20 ma. current loop TTY interface
- Interface for two audio cassette recorders
- Two audio cassette formats: ASCII KIM-1 compatible and binary, blocked file assembler compatible

**ROM RESIDENT ADVANCED INTERACTIVE MONITOR**
Advanced features found only on larger systems . . .
- Monitor-generated prompts
- Single keystroke commands
- Address independent data entry
- Debug aids
- Error messages
- Option and user interface linkage

**ADVANCED INTERACTIVE MONITOR COMMANDS**
- Major Function Entry
- Instruction Entry and Disassembly
- Display/Alter Registers and Memory
- Manipulate Breakpoints
- Control Instruction/Trace
- Control Peripheral Devices
- Call User-Defined Functions
- Comprehensive Text Editor

**LOW COST PLUG-IN OPTIONS**

| | |
|---|---|
| • A65-010—4K Assembler—symbolic, two-pass | **$79.00** |
| • A65-020—8K BASIC Interpreter | **99.00** |
| • 3K RAM Expansion Kit | **50.00** |

**POWER SUPPLY SPECIFICATIONS**
- +5 VDC ± 5% regulated @ 2.0 amps (max)
- +24 VDC ± 15% unregulated @ 2.5 amps (peak) 0.5 amps average

## PRICE: $389.00 (1K RAM)

***Plus*** $4.00 UPS (shipped in U.S. must give ***street*** address), $10 parcel post to APO's, FPO's, Alaska, Hawaii. All international customers write for ordering information.

We manufacture a complete line of high quality expansion boards. Use reader service card to be added to our mailing list, or U.S. residents send $1.00 (International send $3.00 U.S.) for airmail delivery of our complete catalog.

RNB ENTERPRISES INCORPORATED

2951 W. Fairmount Avenue
Phoenix AZ 85017
(602)265-7564

master charge VISA

# 32 K BYTE MEMORY

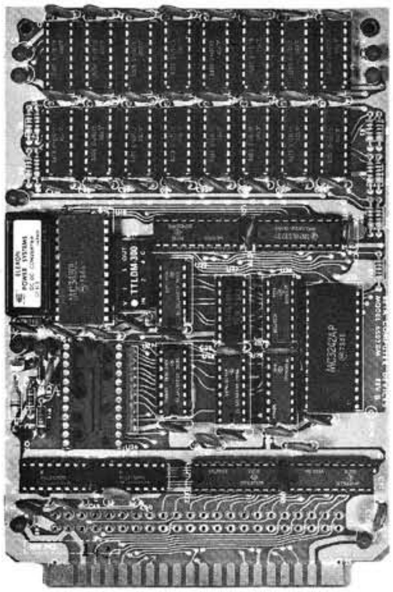
## RELIABLE AND COST EFFECTIVE RAM FOR 6502 & 6800 BASED MICROCOMPUTERS

### AIM 65-*KIM*SYM PET*S44-BUS

- ★ PLUG COMPATIBLE WITH THE AIM-65/SYM EXPANSION CONNECTOR BY USING A RIGHT ANGLE CONNECTOR (SUPPLIED) MOUNTED ON THE BACK OF THE MEMORY BOARD.
- ★ MEMORY BOARD EDGE CONNECTOR PLUGS INTO THE 6800 S 44 BUS.
- ★ CONNECTS TO PET OR KIM USING AN ADAPTOR CABLE.
- ★ RELIABLE—DYNAMIC RAM WITH ON BOARD INVISIBLE REFRESH—LOOKS LIKE STATIC MEMORY BUT AT LOWER COST AND A FRACTION OF THE POWER REQUIRED FOR STATIC BOARDS.
- ★ USES +5V ONLY, SUPPLIED FROM HOST COMPUTER.
- ★ FULL DOCUMENTATION. ASSEMBLED AND TESTED BOARDS ARE GUARANTEED FOR ONE YEAR AND PURCHASE PRICE IS FULLY REFUNDABLE IF BOARD IS RETURNED UNDAMAGED WITHIN 14 DAYS.

| | | |
|---|---|---|
| ASSEMBLED | WITH 32K RAM | $419.00 |
| & | WITH 16K RAM | $349.00 |
| TESTED | WITHOUT RAM CHIPS | $279.00 |
| HARD TO GET PARTS (NO RAM CHIPS) WITH BOARD AND MANUAL | | $109.00 |
| BARE BOARD & MANUAL | | $49.00 |

PET INTERFACE KIT—CONNECTS THE 32K RAM BOARD TO A 4K OR 8K PET. CONTAINS: INTERFACE CABLE, BOARD STANDOFFS, POWER SUPPLY MODIFICATION KIT AND COMPLETE INSTRUCTIONS. .......... $49.00



**16K X 1 DYNAMIC RAM**

THE MK4116-3 IS A 16,384 BIT HIGH SPEED NMOS, DYNAMIC RAM. THEY ARE EQUIVALENT TO THE MOSTEK, TEXAS INSTRUMENTS, OR MOTOROLA 4116-3.

- ★ 200 NSEC ACCESS TIME, 375 NSEC CYCLE TIME.
- ★ 16 PIN TTL COMPATIBLE.
- ★ BURNED IN AND FULLY TESTED.
- ★ PARTS REPLACEMENT GUARANTEED FOR ONE YEAR.

**$8.50 EACH IN QUANTITIES OF 8**

6502 & 6800

**64K BYTE RAM AND CONTROLLER SET**

MAKE 64K BYTE MEMORY FOR YOUR 6800 OR 6502. THIS CHIP SET INCLUDES:

- ★ 32 MSK 4116-3 16KX1, 200 NSEC RAMS.
- ★ 1 MC3480 MEMORY CONTROLLER.
- ★ 1 MC3242A MEMORY ADDRESS MULTIPLEXER AND COUNTER.
- ★ DATA AND APPLICATION SHEETS. PARTS TESTED AND GUARANTEED.

**$295.00 PER SET**

BETA COMPUTER DEVICES
1230 W.COLLINS AVE.
ORANGE, CA 92668
(714) 633-7280

Calif. residents please add 6% sales tax. Mastercharge & Visa accepted. Please allow 14 days for checks to clear bank. Phone orders welcome. Shipping charges will be added to all shipments.

**ALL ASSEMBLED BOARDS AND MEMORY CHIPS CARRY A FULL ONE YEAR REPLACEMENT WARANTY.**

# KIMEX-1 HERE'S A NEAT COMBINATION

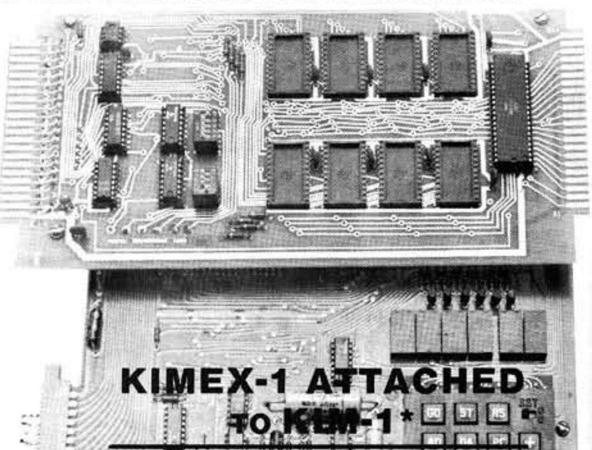
## IDEAL FOR DEDICATED INDUSTRIAL OR PERSONAL APPLICATION

### FEATURES

- PLUGS DIRECTLY INTO AND COVERS UPPER HALF OF KIM-1. EXPANSION FINGERS CARRIED THROUGH FOR FURTHER EXPANSION.
- I/O-POWERFUL 6522 VIA PROVIDED. (VERSATILE INTERFACE ADAPTER) 16 BI-DIRECTIONAL I/O LINES 4 INTERRUPT/HANDSHAKE LINES 2 INTERVAL TIMERS SHIFT REGISTER FOR SERIAL-PARALLEL/PARALLEL-SERIAL OPERATIONS.
- RAM-SOCKETS PROVIDED FOR 4K RAM CONTIGUOUS WITH KIM RAM. (LOW POWER MOSTEK 4118 1KX8's)
- COMPLETE DOCUMENTATION
- EPROM-SOCKETS PROVIDED FOR 8K EPROM. (INTEL 2716 2KX8's)
- BLOCK SELECT SWITCHES FOR EPROM. EPROM USABLE IN ANY ONE OF FOUR 8K BLOCKS FROM 8000H.
- AUTOMATIC RESET ON POWER-UP AND SWITCH SELECTABLE INTERRUPT VECTORS.
- PERMITS UNATTENDED OPERATION.
- LOW POWER CONSUMPTION-5V AT 300 Ma. FULLY LOADED
- BUFFERED ADDRESS LINES
- HIGH QUALITY PC BOARD, SOLDER MASK
- ASSEMBLED AND TESTED

### APPLICATIONS

PROM, RAM AND I/O EXPANSION ON ONE BOARD HAVING MANY INDUSTRIAL/HOME APPLICATIONS FOR DATA ACQUISITION, PROCESS CONTROL, AUTOMATIC CONTROL OF FURNACE, SOLAR HEAT, LIGHTING, APPLIANCES, ETC. . . . .




THIS IS THE COMPLETE KIMEX-1

PA RESIDENTS INCLUDE 6% STATE SALES TAX

**DIGITAL ENGINEERING ASSOCIATES**
P.O.BOX 207 • BETHLEHEM, PA 18016

**$139.95**
LIMITED TIME 1K RAM **FREE!!!**

* KIM IS A REGISTERED TRADEMARK OF MOS TECHNOLOGY, INC.

# Nuts And Volts No. 3
## Address Decoding

Gene Zumchak

An important consideration in microprocessor system design is address decoding. Address lines are decoded from the highest order lines to subdivide memory space into smaller blocks. For example, the top three lines, can be used to divide memory into eight 8K blocks; the top four lines, into sixteen 4K blocks; the high six lines, into sixty-four 1K blocks; etc. There are numerous ways to do the decoding in hardware. If addresses need to be changed often, then dip switches and open collector Exclusive NOR (compare) gates can be used to compare any number of address lines with the selected polarity of that address line. The open collector outputs are wire-ORed together. If any gate is false the output will go low. An additional EX-NOR gate can be wired as an inverted to give a low true output. Figure 1. shows this method used to generate the 1K select for $1000 (000100). If the address decoding is to be permanent, a single 3 to 8 decoder like a 74LS138 can be used to give one, or several 1K, 2K, 4K, or 8K selects. Figure 2. shows a 74LS138 wired to give a select for $1000. Still another method is to use a 4-bit magnitude comparator chip, like the 74LS85. These can be used with switches on one of the word inputs, and also can be cascaded to compare longer words.

For 6502 systems, some users use Ø2 as an input to address decoders. This transfers the strobe action required for writing from the write input to the chip select. It also means than "reads" will be gated with Ø2. The user can get away with this only because the hold time requirement for the 6502 on a read operation is so short. As mentioned in the first column, if Ø2 is seriously delayed via the address decode paths, the strobing action could occur after the write data has already gone away. In general, it is not the best practice to "gate" write data with the strobing signal.

For ROM selects, it is desireable to gate in the R/W signal, so that the ROM select can never go true for a write operation. In the AIM, for example, ROM selects are generated from a 2 to 4 decoder. Ironically, this decoder has a gate input that was grounded instead of using the inverted R/W signal. If a write operation is attempted to ROM area, both the ROM and 6502 will attempt to drive the bus. Fortunately, most chips are designed to take momentary shorting and it is most unlikely that any harm will come to the 6502 or ROM. Still, it is careless design not to consider that writes may be attempted to read-only space.

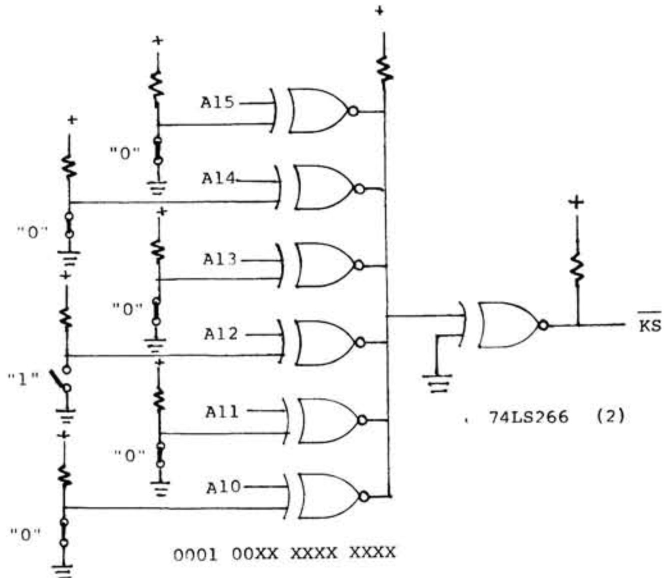



**Fig. 1. 1K Select Using EX-NOR**

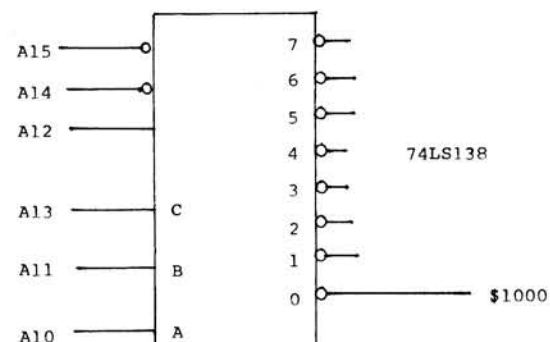



**Fig. 2 1K Select Using 3 to 8 Decoder**

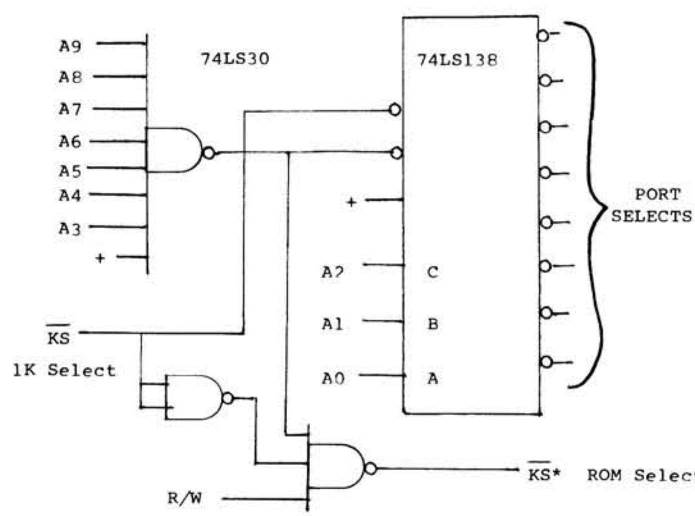



**Fig. 3. Stealing Port Selects.**

### Stealing Addresses

What does one do when a few addresses are needed for I/O ports? Is it necessary to waste a large chunk of space for a few addresses? Clearly, more address lines can be decoded so that decoding is complete.

Still, it is questionable whether that helps much, since if the remaining space is to be used, it will have to be decoded to exclude those addresses. If you have space set aside for ROM, however, and if you won't need every last word, then it is possible to steal a few addresses for I/O without having to waste any other space. Suppose we have a 1K select at $1000 which we will use for ROM. The circuit of figure 3. steals the top eight addresses of a 1K select. The ROM will respond only to 1016 addresses. The top eight addresses generate eight I/O selects with a 74LS138.

When designing 6502 controllers from scratch, the address stealing method illustrated can be used to extract I/O addresses from zero page. Rarely is all, or even half, of zero page needed for scratch pad. Putting I/O in zero page not only can considerably cut program length, but also speed up execution time. In some of my early controller designs, I used a pair of 256 x 4 RAM chips, and decoded addresses so that the RAM straddled pages zero and one. The low half of page zero was thus available for I/O.

**Special 6502 Address Considerations**

All 6502 systems have two address decoding needs in common. First of all, a select must respond to the very highest addresses where the reset and interrupt vectors are located. (It is too bad that one of the unused pins on the 6502 could not have been used for an open drain vector select. This would have allowed the user to wire-OR this select with his own ROM area for response to the reset and interrupts.) The other consideration, of course, is that all 6502 systems need RAM in zero page and page one (for stack).

Interestingly, the KIM, SYM, and AIM use three different methods for interrupt response.The unexpanded KIM avoids the interrupt vector problem by not decoding the top three address lines so that every 8K block is the same. Thus the unexpanded KIM responds to $FFFC at $1FFC. If a KIM is expanded, its address decoder must be enabled only in the lowest 8K block. The interrupt vector area must be decoded with an open collector decoder and wire-ORed to KIM's K7 select ($1C00-$1FFF). The SYM causes the response to the reset vector to be ROM at $8FFC, but the response to the interrupt vectors to be the system RAM at $A600. How does it perform this magic?

The SYM's reset lines go as usual, to the VIA port chips. All I/O lines on the VIA port chips come up in a high state at rest. The CA2 line of one of the VIAs is inverted to give a low-true Power On Reset signal which is effectively wire-ORed to the monitor ROM. Thus upon power up, the SYM finds the reset addresses at $8FFC and $8FFD, not because of decoding, but because the monitor ROM is held fast enabled by the POR line. One of the first things the reset program does is to reset the POR line. This line also inhibits normal address decoding. After POR is reset, normal address decoding takes place, and response to the interrupt vectors will be to the system RAM area (which is preloaded with default interrupt vectors at reset time).

The AIM's solution is the simplest. It merely puts its 8K monitor in the highest 8K block of memory. Thus the reset and vector select is the normal ROM select.

The KIM, SYM, and AIM can all be used as the basis for dedicated controllers. In such an application, it is usually desireable for Reset to cause the user's controller program to begin. For either the KIM or SYM this is no problem. The KIM will require external decoding of the reset vector space. The decode can be wire-ORed to the controller's ROM. The SYM allows the POR signal to be alternatively jumpered to any of the on board ROM sockets. The AIM, unfortunately, will require that some cutting be done to the board, and further decoding tacked on, so that monitor programs can still be used, without the monitor responding to the topmost vector addresses. For any of the three systems mentioned, when using them in a controller application, some input condition should be defined which will cause the program to enter the board's normal monitor programs.

**Summary**

Address decoding uses the highest address lines to subdivide memory space into smaller chunks. Selects are used to enable RAM, ROM and I/O. Address selects can use Ø2 as an input, thus including the strobing action required for writing in the select. ROM selects should include the R/W signal so that they will not respond to write operations. Address stealing can be used to obtain port selects from ROM area without wasting a large chunk of memory space. Similary, I/O addresses may be stolen from zero page RAM area. Various methods can be used to respond to the Reset and interrupt vectors at the top of memory space. A particularly versatile method is the Power On Reset used on the SYM.

I welcome any comments or criticisms you may have of the material in this column. I invite your suggestions for topics for future columns. However, time and space do not allow major design projects for the column. ©

# A Simple Interface For A Stepper Motor

Marvin L. De Jong
Department of Mathematics-Physics
The School of the Ozarks

The circuit shown in Figure 1 and the programs given in Listing 1 allow you to drive a stepper motor with your 6502 based microcomputer. Why run a stepper motor? Perhaps to drive a solar panel to follow the sun, homebrew your own x-y plotter, run a pump at a preselected rate, or turn your robot's head. Whatever your application may be, here is some information to get you started working with stepper motors. You will want to get additional information from the following companies:

AIRPAX
North American Philips Controls Corp.
Chesire Industrial Park
Cheshire, CONN 06410
(203) 272-0301

Dana Industrial
11901 Burke St.
Santa Fe Springs, CA 90670
(213) 698-2595

You can get a nice Stepper Motor Handbook from AIRPAX, and the specification sheet for the Stepper Motor IC Driver SAA1027 also is available from AIRPAX. The circuit we used made use of this integrated circuit driver and an AIRPAX 82701 stepper motor. The circuit was breadboarded on a FIRST MATE/SECOND MATE system from MicroMate.

The circuit of Figure 1 consists of a 7406 inverter with high voltage open-collector outputs. Two pins of the Port B application port on the computer drive the 7406 which in turn controls the trigger (T) input and the rotation direcn (R) input on the stepper motor IC driver. The driver chip controls the stepper motor.

Listing 1 gives several subroutines that may be used to control the motor. The instructions in the INITIALIZE routine should be used near the beginning of any program to drive the stepper motor. These instructions place the proper logic levels on the T and R pins. In Listing 1 the initilization instructions are part of a short program from $0300 to $0328 that will run the stepper motor at a constant rate. The rate used in this program is about 200 steps/second, near the maximum rate for this particular motor. Since each step for the 82701 is a 7.5$^0$ step, the rotation rate is 250 rpm.

The INITIALIZE and MOTOR RUN routines call two subroutines, TRIGGER and either CW or CCW. Calling subroutine TRIGGER produces one step on the stepper motor. If the TRIGGER call is preceded by a subroutine call for CW, then the motor will turn clockwise (CW). If CCW (for counterclock-

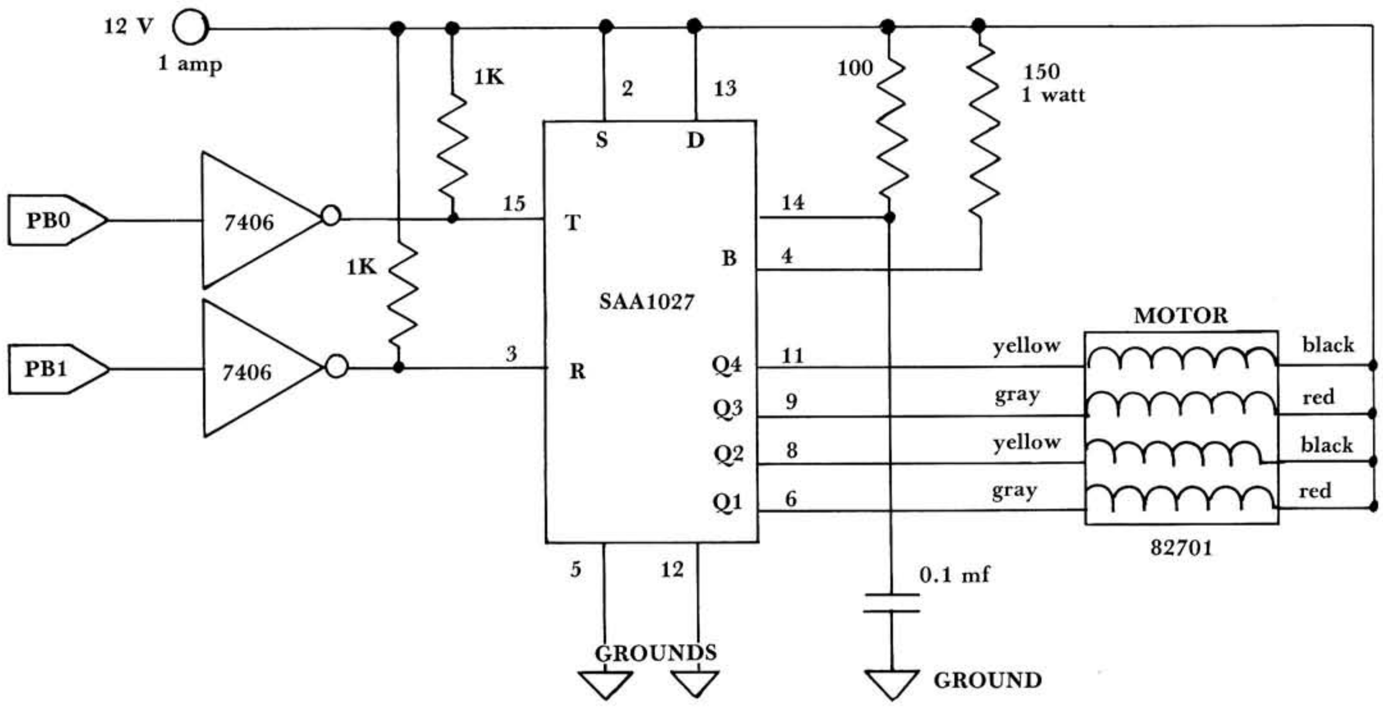



**Figure 1.** Stepper motor interface. The SAA1027 is a special driver integrated circuit. The 7406 will also require five volts for its own power.

wise) is called, then the motor will turn counterclockwise. The MOTOR RUN routine is an infinite loop, and is listed here to show how to make the motor run. Note that we have used the T1 timer on the 6522 VIA to control the time between steps.

Subroutine MOVE can be used to turn the stepper motor a prescribed number of steps, either CW or CCW depending on which subroutine is called. The number of steps is stored in location STEPS at address $0000. Again, the T1 timer on the 6522 VIA is used to produce the necessary delay between steps. The stepper motor is not capable of turning as fast as the computer can toggle the T input, hence either a timer delay or a delay loop must be used to wait.

Be sure to get all the information about various motors and drivers before you get started with your project. Quite obviously, different projects will demand different motors; larger, smaller, geared, linear actuators, etc. Then build something spectacular and let us hear about it.

**Listing 1.** Driver Routines for the Stepper Motor Interface.

```
0300 A9 03     INITIALIZE LDA $03      Set up Port B to make pins PB0 and PB1
0302 8D 02 A0             STA PBDD     output pins.
0305 A9 00                LDA $00      Pull driver pins T and R to logic one
0307 8D 00 A0             STA PBD      through the 7406 inverter.
030A A9 40     MOTOR RUN  LDA $40      Put the T1 timer in its free-running
030C 8D 0B A0             STA ACR      mode by setting bit six to logic one.
030F A9 86                LDA $86      Set up the T1 timer to time out every
0311 8D 04 A0             STA T1LL     5 milliseconds giving 200 steps/sec.
0314 A9 13     MORE       LDA $13      ($1386 + 2 = 5000)
0316 8D 05 A0             STA T1LH     Now the timer is loaded and running.
0319 2C 0D A0  LOAF       BIT IFR      Has the timer timed out?
031C 50 FB                BVC LOAF     No. Then loaf here.
031E 20 07 04             JSR CW       Subroutine CW will result in motor
0321 20 00 04             JSR TRIGGER  running clockwise, CCW turns it counter-
0324 AD 04 A0             LDA T1CL     clockwise. Subroutine TRIGGER produces
0327 18                   CLC          one step of the motor.
0328 90 EA                BCC MORE     Clear the interrupt flag, then return
                                       to make another step.
0400 EE 00 A0  TRIGGER    INC PBD      Pulse the T input of the stepper motor
0403 CE 00 A0             DEC PBD      driver.
0406 60                   RTS
0407 A9 02     CW         LDA $02      Bring the R input to logic zero for
0409 0D 00 A0             ORA PBD      clockwise (CW) rotation, by making
040C 8D 00 A0             STA PBD      PB1 logic one.
040F 60                   RTS
0410 A9 FD     CCW        LDA $FD      Bring the R input to logic one for
0412 2D 00 A0             AND PBD      counterclockwise (CCW) rotation, by
0415 8D 00 A0             STA PBD      making PB1 logic zero.
0418 60                   RTS
0500 A9 00     MOVE       LDA $00      Set up T1 for the one-shot mode
0502 8D 0B A0             STA ACR      by clearing the 6522 ACR.
0505 20 10 04             JSR CCW      Motor will turn counterclockwise.
0508 A9 87                LDA $87      Timer will wait 5 milliseconds between
050A 8D 04 A0             STA T1LL     steps.
050D A9 13     AGAIN      LDA $13
050F 8D 05 A0             STA T1LH     Timer is now loaded and running.
0512 2C 0D A0  WAIT       BIT IFR      Has it timed out?
0515 50 FB                BVC WAIT     No. Then wait here.
0517 20 00 04             JSR TRIGGER  Here the motor turns one step.
051A C6 00                DEC STEPS    Decrement the step counter.
051C D0 EF                BNE AGAIN    Has it reached zero?
051E 60                   RTS          Yes. Then turn is complete.
```

# GET rich QUICK

Gene Zumchak

Actually, it was never my intention to get rich. It's just that I've dropped out of the working world for several months to concentrate on writing a book, and I was starting to miss having a little income. Last summer I read Don Lancaster's "Incredible Secret Money Machine" which is a very entertaining dissertation on going into business for yourself. A couple of his ideas are keep it small, and sell it cheap. It occurred to me that selling some blank PC boards might fit those qualifications. So I asked myself, what does everyone need? My most useful accessory is my EPROM programmer. I don't need it that often, but it's indispensible when I do. With 2708's dropping down to the $7 level, and five volt only 2716's dipping below $20, it would seem that no one should be without an EPROM programmer.

In small quantities I have to pay a little over $6 for a blank board. I figured if I offered the board for $19, I'd make $10 on every one I sold. If I sold a modest fifteen a month, that would make a $150 dent in my office overhead of almost $400. I certainly wouldn't get rich, but it would help a little. It seemed simple enough.

I should have known better. You just can't drop a blank PC board into an envelope and mail it. You need a parts list, schematic, software, circuit description, software description, etc. I already had much of that material, but it required updating, and I ended up doing most of it over from scratch. Before I was finished, I had over 20 pages of documentation, and about $1500 of my time invested. I'll need to sell 150 boards just to pay for my lost time. That is, that's how many I'd have to sell IF I made the $10 a board that I had planned on.

After my ad came out, the inquiries came "pouring" in. Unfortunately, the material I send out is over an ounce and it's costing me $.28 to mail it first class. Then there's the cost of the printed material. In all, it costs over $.50 for every inquiry. If one in twenty (a very high return), results in an order, then it will cost me $10 in mailings to get that order. Well, there goes that $10 profit. Of course, I failed to account for the $10 of my time I spend processing the info requests. Then there's the cost of placing the ad. With a little bit of luck, I'll only lose two or three dollars on every board I sell. Make that five. I forgot to account for some of my overhead. Boy am I dumb.

It sort of looks like, the fewer boards I sell, the better off I am. I guess I should consider myself very fortunate that I've sold only three boards. It does make me wonder, though, how all you non-customers out there program your EPROMs. Maybe you all bought programmers last year. I guess others are waiting for the price to drop.

The very saddest part of this situation is that like other forms of gambling, you don't know how to stop. I reason, I already have $1500 sunk into this, I might as well stick it out a little longer. There's really no hope for me.

Actually, I have been selling KIM accessories for years, so I really ought to know better. For my several thousand hours already invested, I'm only a thousand dollars in the hole. At least I can take comfort in knowing that my lack of success is not attributable to poor quality or design. My customers (both of them) are just thrilled with my work.

If nothing else, I've developed a great deal of respect for those others who are building and selling hardware, and actually making a living at it. If you have designed and built something for yourself that you think the rest of the world could use, forget it quickly, before it's too late. Take a cold shower. If you're thinking of saving a little money by designing and building something yourself, save yourself the misry. Someone else has already done it. Just to duplicate my hardware and software efforts on a simple project like the Programmer would take at least $500 of your time. You're farther ahead using your spare time to sell hamburgers at minimum wage and buying yourself a really nice programmer. But then there's not that satisfaction of doing it yourself.

Actually, the programmer project was the good news. I haven't even told you about the other board, the EPROM Emulator. But that will have to wait. I have to go. There's a couple of gentlemen in white coats at my door. I don't believe it. At last, some customers. ©

# KIM-1 TIDBITS

Harvey B. Herman
Chemistry Department
University of North Carolina at Greensboro
Greensboro, NC 27412

This article is the second in what I hope is a continuing series on the KIM. My intention is to share with others programs which should make KIM easier to use. The reader will please note that the machine language programs have been documented with the excellent Macro Assembler and Text Editor (ASSM/TED) from Eastern House software (see my review in COMPUTE #1, p100). When I started messing with KIM several years ago, my programs were mostly hand assembled. This task has been made much easier by using ASSM/TED.

The first program is an implementation of a KIM real-time clock (sometimes called a tick counter). I have whimsically referred to it as a 'jeffrey counter' after a former student of mine. Every 100 milli-seconds a location-in page zero is incremented. By peeking at this location one can time external events up to about 25 seconds. We have used it to tell when to take readings from an analog-to-digital converter. The clock can be started, stopped or read under program control. A source listing of the program is shown in figure 1. An example of a BASIC program which uses the tick counter is shown in figure 2.

In order for this program to work one external connection needs to be made. The counter is interrupt driven and requires PB7 on the application connector (A - 15) to be connected directly to IRQ (E - 4). The program sets PB7 as an input line and initializes the IRQ vectors at \$17FE/\$17FF to point to the clock service routine. For convenience I have modified KIM Microsoft BASIC to execute a preamble which, among other things, sets up these vectors before jumping to the normal start of BASIC.

The second program is an enhancement to KIM Microsoft BASIC. Support is added for a terminal (e.g. ASR 33 Teletype) which used the X - ON/X - OFF protocol to start and stop a paper tape reader. Over the years I had accumulated a number of programs on paper tape which I wanted to use with KIM BASIC. As supplied the BASIC software did not read paper tapes reliably and I had no desire to key in long programs again. I waded through a dissassembly of BASIC and found two calls to a subroutine which could be called "input a line".

```
            ;
            ;INTERRUPT SERVICE ROUTINE FOR REAL-TIME CLOCK
            ;(TICK COUNTER). ENHANCEMENT TO KIM MICROSOFT
            ;BASIC. 1/10 SEC PER TIC.
            ;
            ;HARVEY B. HERMAN
            ;
            ;REQUIRES CONNECTION OF IRQ TO PB7.
            ;SET ALL PB PINS AS INPUT. SET ORIGINAL COUNT
            ;AND DIVIDE RATE(/1024) OF 6530 TIMER.
            ;START TICKING-POKE 5891,0:POKE 5903,98
            ;DISABLE IRQ(OTHER WAYS POSSIBLE).
            ;STOP TICKING-POKE 5894,98
            ;READ TICK COUNTER-PEEK(224)
            ;RESET STOPS CLOCK.
            ;
            TICK        .DE $E0       ;FREE LOCATION PAGE ZERO
            COUNT       .DE $62       ;1/10 SEC.
            CLKKTE      .DE $170F     ;DIVIDE BY 1024(INT. EN.)
            IRQL        .DE $17FE     ;KIM IRQ INTERRUPT
            IRQH        .DE $17FF     ;    VECTORS
            ;
            ;INITIALIZATION ROUTINE
            ;SET UP VECTORS AND ZERO TICK COUNTER
            ;LOCATE ANYWHERE CONVENIENT
                        .BA $4368
            ;OTHER CODE ABOVE IN MY VERSION
4368- A9 DA             LDA #INTER
436A- 8D FE 17          STA IRQL
436D- A9 02             LDA #H,INTER
436F- 8D FF 17          STA IRQH
4372- A9 00             LDA #00
4374- 85 E0             STA *TICK
            ;OTHER CODE BELOW IN MY VERSION
            ;
            ;INTERRUPT HERE ON TIMEOUT
                        .BA $2DA
02DA- 48    INTER       PHA
02DB- EA                NOP  ;HIDE MY IGNORANCE
02DC- E6 E0             INC *TICK
02DE- A9 62             LDA #COUNT
02E0- 8D 0F 17          STA CLKKTE
02E3- 68                PLA
02E4- 40                RTI
                        .EN
```

# Skyles Electric Works

## Presenting the Skyles MacroTeA

# The Software Development System For the Serious Programmer

## Text Editor

To help you write your program, MacroTeA includes a powerful text editor with **34 command functions:**

| | |
|---|---|
| AUTO | Numbers lines automatically. |
| NUMBER | Automatically renumbers lines. |
| FORMAT | Outputs text file in easy-to-read columns. |
| COPY | Copies a line or group of lines to a new location. |
| MOVE | Moves a line or group of lines to a new location. |
| DELETE | Deletes a line or group of lines. |
| CLEAR | Clears the text file. |
| PRINT | Prints a line or group of lines to the PET screen. |
| PUT | Saves a line or group of lines of text on the tape (or disc). |
| GET | Loads a previously saved line or group of lines of text from the tape (or disc). |
| DUPLICATE | Copies text file modules from one tape recorder to the other. Stops on specific modules to allow changes before it is duplicated. This command makes an unlimited length program (text file) practical. |
| HARD | Prints out text file on printer. |
| ASSEMBLE | Assembles text file with or without a listing. Assembly may be specified for the object code (program) to be recorded or placed in RAM memory. |
| PASS | Does second pass of assembly. Another command that makes unlimited length text files (source code) practical. |
| RUN | Runs (executes) a previously assembled program. |
| SYMBOLS | Prints out the symbol table (label file). |
| SET | Gives complete control of the size and location of the text file (source file), label file (symbol table) and relocatable buffer. |
| DISK | Gives complete access to the eleven DOS commands; PUT GET NEW INITIALIZE DIRECTORY COPY DUPLICATE SCRATCH VALIDATE RENAME ERROR REPORT |
| EDIT | Offers unbelievably powerful search and replace capability. Many large computer assemblers lack this sophistication. |
| FIND | Searches text file for defined strings. Optionally prints them and counts them; i.e., this command counts number of characters in text file. |
| MANUSCRIPT | Eliminates line numbers on PRINT and HARD command. Makes MacroTea a true and powerful Text Editor. |
| BREAK | Breaks to the Monitor portion of MacroTea. A return to Text Editor without loss of text is possible. |
| USER | Improves or tailors MacroTea's Text Editor to user's needs; "Do-it-yourself" command. |

## Fast...Fast Assembler

Briefly, the pseudo-ops are:

| | |
|---|---|
| • BA | Commands the assembler to begin placing assembled code where indicated. |
| • CE | Commands the assembler to continue assembly unless certain serious errors occur. All errors are printed out. |
| • LS | Commands the assembler to start listing source (text file) from this point on. |
| • LC | Commands the assembler to stop list source (text file) from this point in the program. |
| • CT | Commands the assembler to continue that source program (text file) on tape. |
| • OS | Commands the assembler to store the object code in memory. |
| • OC | Commands the assembler to not store object code in memory. |
| • MC | Commands the assembler to store object code at location different from the location in which it is assembling object code. |
| • SE | Commands the assembler to store an external address. |
| • DS | Commands the assembler to set aside a block of storage. |
| • BY | Commands the assembler to store data. |
| • SI | Commands the assembler to store an internal address. |
| • DE | Commands the assembler to calculate an external label expression. |
| • DI | Commands the assembler to calculate an internal label expression. |
| • EN | Informs the assembler that this is the end of the program. |
| • EJ | Commands the assembler to eject to top of page on printer copy. |
| • SET | A directive not a pseudo-op, directs the assemblers to redefine the value of a label. |

## Macro Assembler

The macro pseudo-ops include:

| | |
|---|---|
| MD | This is a macro beginning instruction definition. |
| ME | This is end of a macro instruction definition. |
| EC | Do not output macro-generated code in source listing. |
| ES | Do output macro-generated code in source listing. |

## Conditional Assembler

The conditional assembly pseudo-ops are:

| | |
|---|---|
| IEQ | If the label expression is equal to zero, assemble this block of source code (text file). |
| INE | If the label expression is not equal to zero, assemble this block of source code (text file). |
| IPL | If the label expression is positive, assemble this block of source code. |
| IMI | If the label expression is negative, assemble this block of source code. |
| ••• | This is the end of a block of source code. |

## Enhanced Monitor

*. . . By having 16 powerful commands:*

| | |
|---|---|
| A | Automatic MacroTeA cold start from Monitor. |
| Z | Automatic MacroTeA warm start from Monitor. |
| F | Loads from tape object code program. |
| S | Saves to tape object code between locations specified. |
| D | Disassembles object code back to source listing. |
| M | Displays in memory object code starting at selected location. The normal PET screen edit may be used to change the object code. |
| R | Displays in register. Contents may be changed using PET screen edit capabilities. |
| H | Hunts memory for a particular group of object codes. |
| W | Allows you to walk through the program one step at a time. |
| B | Breakpoint to occur after specified number of passes past specified address. |
| Q | Start on specified address. Quit if STOP key or breakpoint occurs. |
| T | Transfers a program or part of a program from one memory area to another. |
| G | Go!! Runs machine language program starting at selected location. |
| X | Exits back to BASIC. |
| I | Display memory and decoded ASCII characters. |
| P | Pack (fill) memory with specified byte. |

## What are the other unique features of the MacroTeA?

- Labels up to 10 characters in length
- 50 different symbols to choose from for each character
- $10^{16}$ different labels possible
- Create executable object code in memory or store on tape
- Text editor may be used for composing letters, manuscripts, etc.
- Text may be loaded and stored from tape or disc
- Powerful two-cassette duplicator function
- String search capability
- Macros may be nested 32 deep
- 25 Assembler psuedo-ops
- 5 Conditional assembler psuedo-ops
- 40 Error codes to pinpoint problems
- 16 Error codes related to Macros
- Warm-start button
- Enhanced monitor with 16 commands

Truly, there is simply no other system of this magnitude at anywhere near this price. **$395.00 ***

*(With any Skyles Memory Expansion System,* **$375.00***)*

*California residents please add 6% or 6.5% sales tax as required*

VISA, MASTERCHARGE ORDERS CALL **(800) 538-3083** (except California residents)

CALIFORNIA ORDERS PLEASE CALL **(408) 257-9140**

**Skyles Electric Works**

231 E South Whisman Road
Mountain View, CA 94041
(415) 965-1735

The change I made was to send out an X - ON character (with added code in page 2) before jumping to this subroutine. My paper tape reader will begin reading upon receipt of this character.

It was necessary to make one further change in order for KIM to punch BASIC program tapes that would read back properly. The tape reader must stop after carriage return while BASIC "digests" a line. Therefore the X - OFF character (stop reading) must be included in the data stream. I found a call to BASIC's output routine immediately after loading the accumulator with "CR" (hex 0D). I changed this to a jump to code, again in page 2, which outputs X - OFF and CR before continuing as before. Later when the teletype reads the X - OFF character on tape it shuts off but does not stop immediately and reads one additional character (CR). Return sets the BASIC interpreter off and running. In a short time, after digesting the line, BASIC sends the X - ON character asking for more data. Only one *caveat* - remember to type control/0 before and after reading tapes so the teletype will only single space.

I hope these additional programs will be found useful. As always, if you have any questions I will be happy to respond if you include a SASE.

```
10 REM EXAMPLE PROGRAM TO PRINT A NUMBER
20 REM ONCE EVERY 10 SECONDS
30 POKE 5903,98: REM START CLOCK
40 FOR I = 1 TO 10
50 POKE 224,0: REM ZERO CLOCK
60 IF PEEK (224) <100 THEN 60
70 PRINT I;
80 NEXT I
90 POKE 5894,98: REM STOP CLOCK
100 END
```

```
                  ;
                  ;X-ON/X-OFF ENHANCEMENT TO
                  ;KIM MICROSOFT BASIC
                  ;SERIAL NUMBER 9011
                  ;
                  ;HARVEY B. HERMAN
                  ;
                  ;SENDS X-ON(HEX 11) TO TERMINAL BEFORE JUMPING TO
                  ;"INPUT A LINE". TERMINALS WITH THIS FEATURE WILL
                  ;AUTOMATICALLY START READING PAPER TAPE. WHEN AN
                  ;X-OFF(HEX 13) IS READ, THE TAPE READER CONTROL
                  ;WILL TURN OFF AND THE READER WILL COAST AND
                  ;TRANSMIT ONE EXTRA CHARACTER.
                  ;
                  ;PUNCHES PAPER TAPE(BY LIST) WITH X-OFF/CR/LF/
                  ;NULL(S) AS END OF LINE FORMAT.
                  ;
                  ;NULL(S) CORRECTION(NECESSARY FOR EARLY VERSIONS
                  ;OF BASIC).
                  ;
                  OUTCH      .DE $1EA0     ;KIM OUTPUT ROUTINE
                  INPUT      .DE $2426     ;BASIC "INPUT A LINE"
                  OUTPUT     .DE $2A3A     ;BASIC OUTPUT ROUTINE
                  LDA0       .DE $29D1     ;INSTRUCTION LDA #00
                  ;
                  ;INTERCEPT CALLS TO "INPUT A LINE"
                             .BA $2351
2351- 20 C8 02               JSR XON
                             .BA $2AB6
2AB6- 4C C8 02               JMP XON
                  ;OUTPUT X-ON CHARACTER
                             .BA $2C8
02C8- A9 11       XON        LDA #$11      ;X-ON
02CA- 20 A0 1E               JSR OUTCH     ;OUT TO TERMINAL
02CD- 4C 26 24               JMP INPUT     ;INPUT A LINE
                  ;INTERCEPT CALLS TO OUTPUT CR
                             .BA $29C3
29C3- 20 D0 02               JSR XOFF
                  ;OUTPUT X-OFF/CR CHARACTERS
                             .BA $2D0
02D0- A9 13       XOFF       LDA #$13      ;X-OFF
02D2- 20 3A 2A               JSR OUTPUT    ;BASIC OUTPUT ROUTINE
02D5- A9 0D                  LDA #$0D      ;CR
02D7- 4C 3A 2A               JMP OUTPUT
                  ;CORRECT NULL(S) ERROR IN EARLY VERSION
                  ;OF BASIC. A WAS DESTROYED BY OUTPUT AND MUST
                  ;BE LOADED AGAIN WITH ZERO.
                             .BA $29D7
29D7- D0 F8                  BNE LDA0
                             .EN
```

©

# SYM-1 Home Warning System

A. M. MacKay
600 Sixth Avenue West
Owen Sound, Ontario
N4K 5E7

If you have a C.R.T. hooked up to your SYM-1, this program and a couple of dollars worth of parts will let you experiment with a home warning system, and may help you learn a bit about your computer.

Once the hardware shown in figures 1 to 3 is connected, load the program and hit "RUN START" if you have an assembler or G 200 CR if you don't. Then as long as the old homestead is devoid of marauders, catastrophes and other unthinkables your screen will steadily and quietly tell you that everything is O.K. However, if one or more of the sensors detects any of the aforementioned nasties, a siren sounds and a message flashes on the screen telling you the nature and location of the problem(s).

The program starts by clearing the screen and displaying the "EVERYTHING IS O.K." message. It then polls the sensors, one at a time, beginning with PB0. As soon as it finds an active sensor, it clears the "O.K." message and displays the correct warning, then checks the remaining sensors. When all sensors have been checked and messages have been displayed for all active sensors, the siren sounds once and the polling process is repeated. The warnings and the siren will continue until all the sensors are turned off, at which time the "EVERYTHING IS O.K." message re-appears and polling of the sensors resumes.

For this experiment four switches are used as input sensors. In practice,

```
;         *********************************
;         *********************************
;         ***                           ***
;         ***    HOME WARNING SYSTEM    ***
;         ***    FOR  SYM-1 COMPUTER    ***
;         ***                           ***
;         ***    BY  A. M. MACKAY       ***
;         ***    600 SIXTH AVE. WEST    ***
;         ***    OWEN SOUND, ONTARIO    ***
;         ***    CANADA        N4K 5E7  ***
;         ***                           ***
;         *********************************
;         *********************************
;
;
;            * * * DEFINITIONS   * * *
;
STATUS       .DE $AC00
OUTVEC       .DE $A663
;
;            * * * INITIATE * * *
;
START        LDA #$F0       ;SET DDRB
             LDA STATUS+2 :   FOR INPUT
             LDA #$00       ;TURN OFF
             STA STATUS     ;   SPEAKER
;
;            * * * SENSOR POLL ROUTINE * * *
;
CLEAR        JMP OK         ;PRINT "ALL O.K." MESSAGE
POLL         LDA STATUS     ;LOOK AT SENSORS
             AND #$0F       ;  IF ONE IS ON
             BNE TEST1      ;     GO TO TEST1
             JMP POLL       ;ELSE GO TO POLL
;
;            * * * SIGNAL PROCESSING ROUTINE * * *
;
TEST1        JSR POSIT      ;POSITION MESSAGE ON SCREEN
             LDA STATUS     ;LOOK AT SENSORS
             AND #$01       ;MASK OFF ALL BUT 1ST
             BEQ TEST2      ;NOT ON? GO TO TEST2
             JSR SPACE      ;ON? PROCESS SIGNAL
             LDX #0         ;INDEX FOR MESSAGE #1
MESS1        LDA TAB1,X     ;GET CHARACTERS
             JSR OUTVEC     ;WRITE ON CRT
             INX  ;         ;NEXT CHARACTER
             CMP #$00       ;LAST ONE?
             BNE MESS1      ;NO? GET NEXT
TEST2        LDA STATUS     ;YES? TEST FOR
             AND #$02       ;  SENSOR #2
             BEQ TEST3      ;     ETC.
             JSR SPACE
             LDX #0
MESS2        LDA TAB2,X
             JSR OUTVEC
             INX
             CMP #$00
             BNE MESS2
TEST3        LDA STATUS     ;TEST FOR
             AND #$04       ;  SENSOR #3
             BEQ TEST4      ;     ETC.
             JSR SPACE
             LDX #0
MESS3        LDA TAB3,X
             JSR OUTVEC
             INX
             CMP #$00
             BNE MESS3
TEST4        LDA STATUS     ;TEST FOR
```

any sensor that will put out 5V under the appropriate conditions will do the job, and the program can be extended to handle any reasonable number of sensors.

The program as written uses PB0 to PB3 of U29 as inputs, and PB7 with its buffer as the speaker output. Figure 1 shows how to hook up buffer B7 for this application. Although it looks complicated, changing B7 is easy. All you have to do is remove one jumper wire and add two resistors and one diode. However, if you prefer, you can forget about buffer B7 and construct a similar speaker system externally. Again, in real life, PB7 would be connected to a large amplifier and speaker.

Figure 2 shows how to attach the speaker, and figure 3 shows one way to use switches as simulated sensors. The speaker volume control, VR1, is optional, but it's a good idea to use it because without it the siren can shatter your wife's teeth.

The messages should, of course, be changed to suit your particular application. If you have an assembler such as RAE, this is easy. If not, you will have to get an ASCII code table and substitute the message code as required. The messages shown in the listing are for a 64x16 screen, so if yours is different, change the number of "0A"'s and "20"'s to suit your requirements.

Sound is produced by toggling the speaker on and off, with a delay between the toggles. The length of the delay determines the frequency. The siren sound is produced by shortening the delay slightly each time the speaker toggles, thus giving a steadily increasing frequency. For more information on sirens and other experiments read Rodnay Zak's "6502 Application Book" published by Sybex.

This program uses the SYM-1 as a dedicated con troller, so it can't be used for other purposes while the program is running. If you want the warning system working while you use your SYM-1 for other things, a new program using interrupts must be written. But that's another ballgame.

```
                AND #$08        ;  SENSOR #4
                BNE LAST        ;    ETC.
                JMP MESSX
LAST            JSR SPACE
                LDX #0
MESS4           LDA TAB4,X
                JSR OUTVEC
                INX
                CMP #$00
                BNE MESS4
                JMP SIREN
MESSX           LDA STATUS      ;TEST FOR
                AND #$0F        ;  ANY SENSORS
                BEQ AGAIN       ;IF NONE, BACK TO POLL
                JMP SIREN       ;ELSE SOUND SIREN
OK              JSR POSIT       ;POSITION
AGAIN           LDX #0          ;  MESSAGE
ALLOK           LDA TAB5,X      ;DISPLAY
                JSR OUTVEC      ;  "EVERYTHING O.K."
                INX  ;               MESSAGE
                CMP #$00
                BNE ALLOK
                LDA #$00        ;TURN OFF
                STA STATUS      ;  SPEAKER AND
                JMP POLL        ;    RESUME POLLING
;
;               * * * SIREN ROUTINE * * *
;
SIREN           LDA STATUS      ;ANY SENSOR
                AND #$0F        ;  STILL ON?
                STA *$CB        ;IF YES, STORE IT
                BNE SCREAM      ;  AND START ALARM
                JMP CLEAR       ;ELSE POLL AGAIN
SCREAM          LDA #$68        ;FREQUENCY CONSTANT
                STA *$CA        ;  AT LOCATION $CA
YLOOP           LDY #$07        ;DELAY CONSTANT
SHRIEK          JSR SPKR        ;TOGGLE SPEAKER
                DEY
                BNE SHRIEK
                INC *$CA        ;INCREMENT
                LDA *$CA        ;  FREQ. CONSTANT
                CMP #$B0        ;HIGHEST CONST.= $B0
                BNE YLOOP
                LDA STATUS      ;ANY
                AND #$0F        ;  SENSOR
                CMP *$CB        ;    CHANGE?
                BEQ SCREAM      ;NO? KEEP SIREN GOING
                LDA STATUS      ;YES? ANY MORE
                AND #$0F        ;  SENSORS ON?
                BEQ SCREEN      ;NO? PRINT O.K. MESSAGE
                JMP POLL        ;YES? PROCESS AGAIN
SCREEN          JMP CLEAR
;
;               * * * SUBROUTINES * * *
;
SPKR            LDA #$80        ;RESET
                STA STATUS+2    :  DDRB AND
                STA STATUS      ;    TOGGLE SPEAKER
                JSR DELAY       ;WAIT AND
                LDA #$00        ;  TOGGLE
                STA STATUS      ;    AGAIN
                JSR DELAY       ;WAIT AGAIN
                RTS
DELAY           LDX *$CA        ;CHANGE
XLOOP           INX  ;            FREQUENCY
                CPX #$00
                BNE XLOOP
                RTS
POSIT           LDX #0
```

```
MOVE        LDA TAB6,X    ;POSITION
            JSR OUTVEC    ;  MESSAGE
            INX  ;             ON
            CMP #$00      ;       SCREEN
            BNE MOVE
            RTS
SPACE       LDX #0        ;POSITION
RIGHT       LDA TAB7,X    ;  WARNING
            JSR OUTVEC    ;     MESSAGES
            INX
            CMP #$00
            BNE RIGHT
            RTS
;
;           * * * MESSAGES * * *
;
TAB1        .BY '[ [ [  WATER IN BASEMENT  ] ] ]' $00
TAB2        .BY '+ + + DOG NEEDS TO GO OUT + + +' $00
TAB3        .BY '# # # THIEF IN WINE CELLAR # # #' $00
TAB4        .BY '* * * MILKMAN IN WIFE"S BEDROOM '
            .BY '* * *' $00
TAB5        .BY '********************************'
            .BY '********' $0D $0A $0A
            .BY '* * * SYM-1 HOME WARNING SYSTEM '
            .BY '* * *' $0D $0A $0A
            .BY '> > > >  EVERYTHING IS O.K.  < '
            .BY '< < <' $0D $0A $0A '********'
            .BY '********************************' $00
TAB6        .BY $0C $0D $0A $0A $0A $0A $00
TAB7        .BY $0D $0A $0A $20 $20 $20 $20 $20
            .BY $20 $20 $00
            .EN
//
```

**Parts List For SYM-1 Home Warning System.**

**R4 100K Resistor**
**R17 10 Ohm Resistor**
**R18 1K Resistor**
**VR1 100 Ohm Potentiometer (Optional)**
**C1 0.01 UF Capacitor**
**CR4 Diode, 1N4148, 1N914 Or Equivalent**
**SW1-4 Any SPDT Switches**
**SPKI 8 Ohm Speaker, RS 40 -247 Or Equivalent**

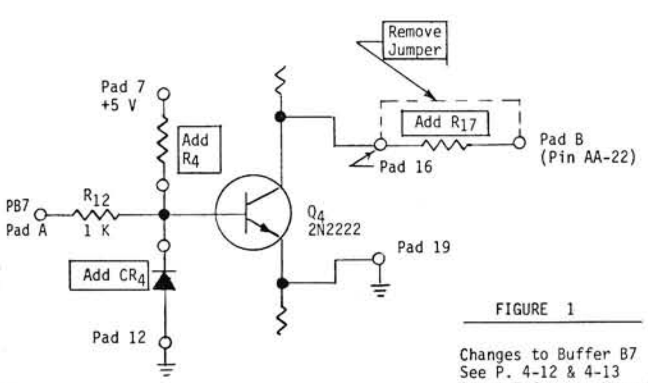



FIGURE 1

Changes to Buffer B7
See P. 4-12 & 4-13
Sym-1 Reference Manual

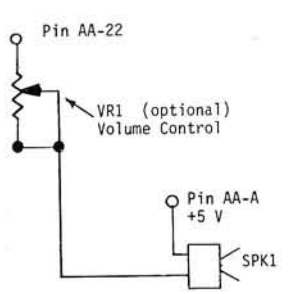



FIGURE 2
Speaker Connections

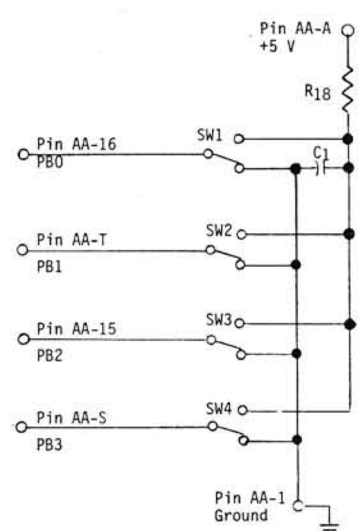
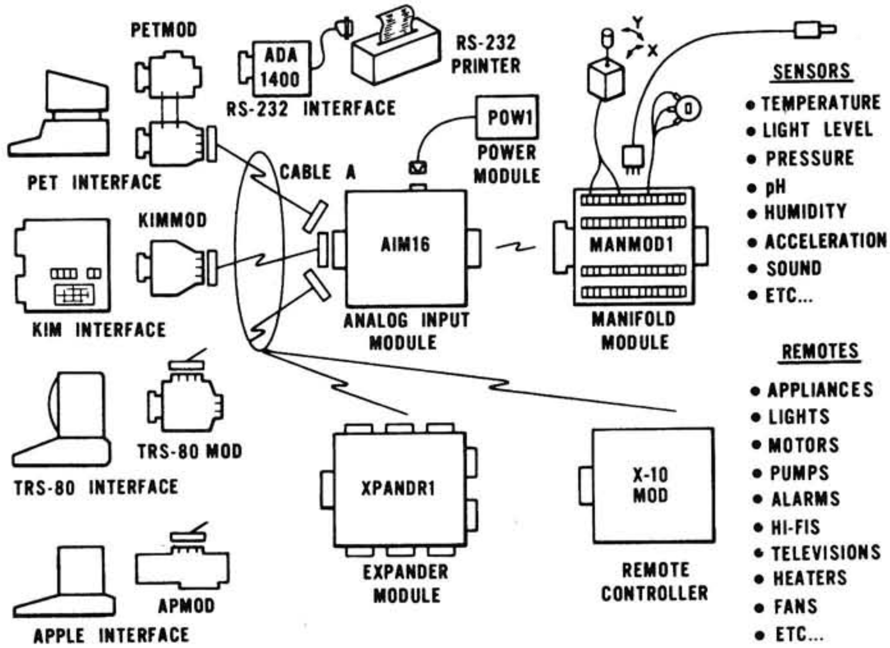



FIGURE 3
Switch (sensor) Connections

# MICROCOMPUTER MEASUREMENT and



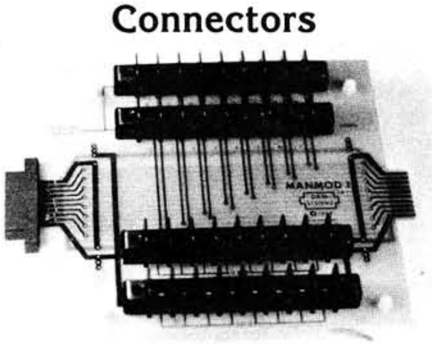

The world we live in is full of variables we want to measure. These include weight, temperature, pressure, humidity, speed and fluid level. These variables are continuous and their values may be represented by a voltage. This voltage is the analog of the physical variable. A device which converts a physical, mechanical or chemical quantity to a voltage is called a sensor.

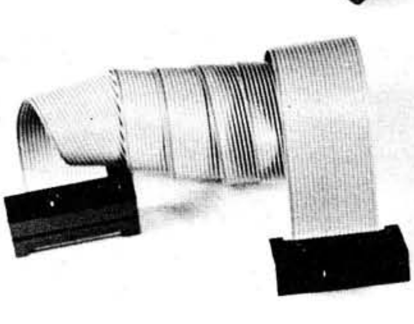
Computers do not understand voltages: They understand bits. Bits are digital signals. A device which converts voltages to bits is an analog-to-digital converter.

Our AIM 16 (Analog Input Module) is a 16 input analog-to-digital converter.

The goal of Connecticut microComputer in designing the uMAC SYSTEMS is to produce easy to use, low cost data acquisition and control modules for small computers. These acquisition and control modules will include digital input sensing (e.g. switches), analog input sensing (e.g. temperature, humidity), digital output control (e.g. lamps, motors, alarms), and analog output control (e.g. X-Y plotters, or oscilloscopes).

## Connectors



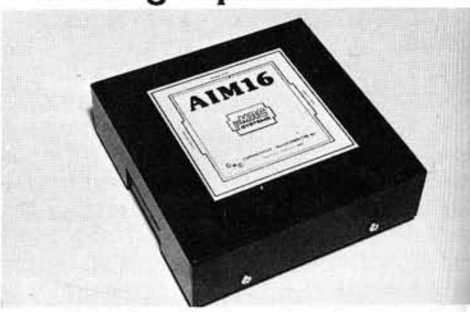



The AIM 16 requires connections to its input port (analog inputs) and its output port (computer interface). The ICON (Input CONnector) is a 20 pin, solder eyelet, edge connector for connecting inputs to each of the AIM16's 16 channels. The OCON (Output CONnector) is a 20 pin, solder eyelet edge connector for connecting the computer's input and output ports to the AIM16.

The MANMOD1 (MANifold MODule) replaces the ICON. It has screw terminals and barrier strips for all 16 inputs for connecting pots, joysticks, voltage sources, etc.

CABLE A24 (24 inch interconnect cable) has an interface connector on one end and an OCON equivalent on the other. This cable provides connections between the uMACSYSTEMS computer interfaces and the AIM 16 or XPANDR1 and between the XPANDR1 and up to eight AIM 16s.

## Analog Input Module




The AIM 16 is a 16 channel analog to digital converter designed to work with most microcomputers. The AIM16 is connected to the host computer through the computer's 8 bit input port and 8 bit output port, or through one of the uMAC SYSTEMS special interfaces.

The input voltage range is 0 to 5.12 volts. The input voltage is converted to a count between 0 and 255 (00 and FF hex). Resolution is 20 millivolts per count. Accuracy is 0.5% ± 1 bit. Conversion time is less than 100 microseconds per channel. All 16 channels can be scanned in less than 1.5 milliseconds.

Power requirements are 12 volts DC at 60 ma.

The POW1 is the power module for the AIM16. One POW1 supplies enough power for one AIM16, one MANMOD1, sixteen sensors, one XPANDR1 and one computer interface. The POW1 comes in an American version (POW1a) for 110 VAC and in a European version (POW1e) for 230 VAC.

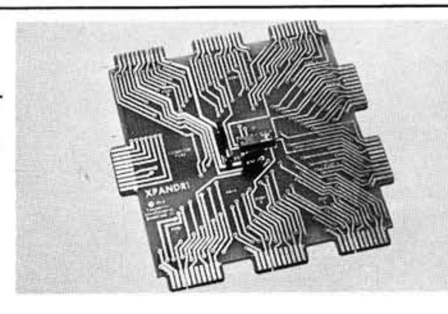
## TEMPSENS


This module provides two temperature probes for use by the AIM16. This module should be used with the MANMOD1 for ease of hookup. The MANMOD1 will support up to 16 probes (eight TEMPSENS modules).

Resolution for each probe is 1°F.

## XPANDR1

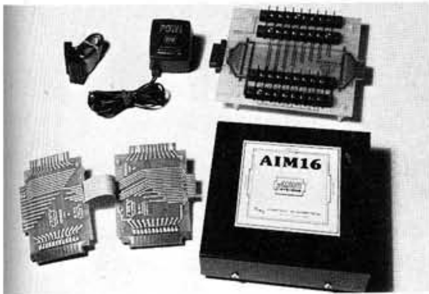
The XPANDR1 allows up to eight Input/Output modules to be connected to a computer at one time. The XPANDR1 is connected to the computer in place of the AIM16. Up to eight AIM16 modules are then connected to each of the eight ports provided using a CABLE A24 for each module. Power for the XPANDR1 is derived from the AIM16 connected to the first port.



# CONTROL for PET, Apple, KIM, and AIM

μMAC SYSTEMS

## Computer Interfaces and Sets



For your convenience the AIM16 comes as part of a number of sets. The minimum configuration for a usable system is the AIM16, one POW1, one ICON and one OCON. The AIM16 Starter Set 2 includes a MANMOD1 in place of the ICON. Both of these sets require that you have a hardware knowledge of your computer and of computer interfacing.

For simple plug compatible systems we also offer computer interfaces and sets for several home computers.

INTRODUCING
SUPER X-10 MODULE

Open a door or window and turn on a light, tape recorder, alarm!

Control lab equipment. CLOSE THE LOOP on the real world.

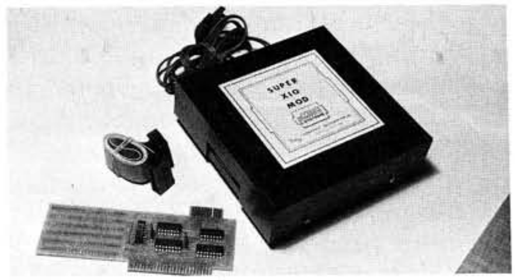
AN INEXPENSIVE CONTROL SOLUTION FOR

HOME SECURITY
ENERGY CONSERVATION
GREENHOUSES
ENVIRONMENTAL CONTROL
INDUSTRIAL CONTROL
LABORATORIES

SUPER X-10 MOD SPECS

1. Remote controller
   Controls up to 256 different remote devices by sending signals over the house wiring to remote modules. Uses BSR remote modules available all over the USA (Sears, Radio Shack, etc.).
   Does not require BSR control module.
   Does not use sonic link.
2. Clock/calendar
   Time of day - hours, minutes, seconds
   Date - month, day - automatically corrects for 28,29,30 and 31 day months.
   Day of the week.
3. Digital inputs/outputs
   8 inputs - TTL levels or switch closures. Can be used as triggers for stored sequences.
   8 outputs - TTL levels


PLUS: CLOCK, CALENDAR, REMOTE SEQUENCE TRIGGERS



4. Computer interfaces
   S-100: Requires one 8-bit input port and one 8-bit output port.
   Requires cable assembly.
   PET, APPLE, TRS-80, KIM, SYM, AIM65: Plug-in sets available - no cable assembly required.
   Other: same as S-100
5. Self-contained module in metal case with its own power supply. Physical size approximately 5X6X2.

Price (until April 30, 1980): $199.00 (S-100), $249.00 (other)

All prices and specifications subject to change without notice. Our 30-day money back guarantee applies.

| | |
|---|---|
| **AIM16 (16 channel-8 bit Analog Input Module)** | **179.00** |
| **POW1a (POWer module-110 VAC)** | **14.95** |
| **POW1e (POWer module-230 VAC)** | **24.95** |
| **ICON (Input CONnector)** | **9.95** |
| **OCON (Output CONnector)** | **9.95** |
| **MANMOD1 (MANifold MODule)** | **59.95** |
| **CABLE A24 (24 inch interconnect cable)** | **19.95** |
| **XPANDR1 (allows up to 8 Input or Output modules to be connected to a computer at one time)** | **59.95** |
| **TEMPSENS2P1 (two temperature probes, -10°F to 160°F)** | **49.95** |
| **LIGHTSENS1P1 (light level probe)** | **59.95** |

**The following sets include one AIM16, one POW1, one OCON and one ICON.**

| | |
|---|---|
| **AIM16 Starter Set 1a (110 VAC)** | **189.00** |
| **AIM16 Starter Set 1e (230 VAC)** | **199.00** |

**The following sets include one AIM16, one POW1, one OCON and one MANMOD1.**

| | |
|---|---|
| **AIM16 Starter Set 2a (110 VAC)** | **239.00** |
| **AIM16 Starter Set 2e (230 VAC)** | **249.00** |

**The following modules plug into their respective computers and, when used with a CABLE A24, eliminate the need for custom wiring of the computer interface.**

| | |
|---|---|
| **PETMOD (Commodore PET)** | **49.95** |
| **KIMMOD (KIM, SYM, AIM65)** | **39.95** |
| **APMOD (APPLE II)** | **59.95** |
| **TRS-80 MOD (Radio Shack TRS-80)** | **59.95** |

μMAC SYSTEMS

## Order Form

**CONNECTICUT microCOMPUTER, Inc.**
***150 POCONO ROAD***
***BROOKFIELD, CONNECTICUT 06804***
*TEL: (203) 775-9659 TWX: 710-456-0052*

**The following sets include one AIM16, one POW1, one MANMOD1, one CABLE A24 and one computer interface module**

| | |
|---|---|
| **PETSET1a (Commodore PET - 110 VAC)** | **295.00** |
| **PETSET1e (Commodore PET - 230 VAC)** | **305.00** |
| **KIMSET1a (KIM, SYM, AIM65 - 110 VAC)** | **285.00** |
| **KIMSET1e (KIM, SYM, AIM65 - 230 VAC)** | **295.00** |
| **APSET1a (APPLE II - 110 VAC)** | **295.00** |
| **APSET1e (APPLE II - 230 VAC)** | **305.00** |
| **TRS-80 SET1a (Radio Shack TRS-80 - 110 VAC)** | **295.00** |
| **TRS-80 SET1e (Radio Shack TRS-80 - 230 VAC)** | **305.00** |

| QUANTITY | DESCRIPTION | PRICE | TOTAL |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| SUBTOTAL | | | |
| Handling and shipping — add per order | | | $3.00 |
| Foreign orders add 10% for AIR postage | | | |
| add 7% sales tax | | | |
| TOTAL ENCLOSED | | | |

NAME ______
COMPANY ______
ADDRESS ______
CITY ______
STATE ______ ZIP ______
VISA ○ M/C ○ Expiration date ______
Card number ______

# A Digital Cardiotachometer Implemented With The AIM 65

Marvin L. De Jong
Department of
Mathematics-Physics
The School of the Ozarks
Pt. Lookout, MO 65726

The circuit shown in Figure 1 and the computer program given in the listing may be used to measure the pulse (heartbeat) rate of a person and display this result (in heartbeats per minute) on the AIM 65 display. A modified version of a PASCO Photo-Plethysmograph (PASCO Scientific, 1933 Republic Avenue, San Leandro, CA 94577) was used to detect the pulses. You might be able to build your own photo-plethysmograph using suitable infared photodiodes and phototransistors (such as the Fairchild FPA106 array) for a lot less money, but be prepared to do some experimentation. The voltage fluctuation from the plethysmograph is amplified by an instrumentation amplifier. We used an Analog Devices (Route 1, Industrial Park, P.O. Box 280, Norwood, MA 02062) AD521. The negative pulses from the AD521 are fed to a 555 timer that acts as a Schmitt trigger circuit, producing a well-defined square pulse at the clock input of the 74LS74. The LED will turn off when a pulse is detected. In the circuit of Figure 1, the 2000 ohm potentiometer on the AD521 controls the gain, and it should be adjusted so that heartbeats cause the LED to flash. An osciolloscope is very useful for making circuit adjustments.

The time interval between two successive pulses to the clock input of the 74LS74 is measured by the computer, and this result is converted to heartbeats per minute. The T1 timer on the 6522 on the AIM 65 microcomputer is used to produce a train of pulses that are fed to one input of the 74LS00 NAND gate. the period of these pulses is 100 microseconds. The leading edge of the heartbeat pulse at the clock input of the 74LS74 flips the Q output to logic one, gating the pulses from PB7 onto PB6 where the T2 counter/timer on the 6522 counts them. They are counted until the leading edge of the next heartbeat pulse flops the Q output to logic zero, closing the gate. Thus, the computer contains the number of 100 microsecond pulses that occurred between two heartbeats. Since $f = 1/T$ where f is the frequency of the heartbeats and T is the time interval between beats, then $f = 10^4/N$ where N is the number of pulses counted by the T2 counter/timer. Changing the units to pulses per minute gives $f = 60 \times 10^4/N$.

We first describe the machine language subroutine called by the BASIC program. The instructions from $0E00 to $0E30 merely initialize the various 6522 registers. For example, T1 must produce a pulse train on PB7 and T2 must count pulses. A positive transition on CB1 must set a flag, while reading Port B produces a one microsecond pulse on CB2, clearing the flip-flop. Starting with the instruction at $0E30, the D-input of the flip-flop is set at logic one. Since the flip-flop was previously cleared, its output is currently at logic zero on the Q pin. The first heartbeat pulse reaching the flip-flop clock input sets the Q output to logic one. This transition is also detected by the instructions at $0E37 through $0E39, and as soon as it occurs, the program begins to watch the pulse count on T2.

Before switching the D-input of the flip-flop to logic zero in order to turn the gate off when the next heartbeat pulse arrives, the T2 counter is watched to allow approximately one-half a heartbeat period to elapse. It waits 0.0244 s to be exact (you may wish to decrease the value of the byte at $0E44). The reason for waiting lies in the fact that the waveshape of the heartbeat pulse (at lease mine) has a secondary peek that can trigger the flip-flop if the gain is set a bit too high. So, we wait until this secondary pulse has been completed before catching the next heartbeat pulse. Bringing the D-input to logic zero allows the next clock pulse to switch Q to logic zero, closing the gate. The number of counts in T2 is obtained, and if T2 counted through zero, producing an interrupt (IRQ) request, the number of interrupts (counts exceeding 65536) are also obtained.

The BASIC program calls the machine language subroutine which then measures the number of 100 microsecond intervals between heartbeats. The BASIC program merely converts this number to a frequency and displays the result. Finally, it returns to measure another heartbeat period. Note that because of various time delays in processing the data, only every other heartbeat interval is measured.

I would not recommend that a novice experimenter with very little test equipment attempt to build this circuit. A number of adjustments are necessary and things like fluorescent lights can cause problems with the plethysmograph, in particular they can produce a 120 cycle modulation that triggers the circuit. The circuit does make a nice electronics project, and I am sure that improvements are possible. Of course, the circuit and the program are meant to be used for instructional purposes rather than in actual medical applications.

Besides being useful in teaching some simple biomedical instrumentation, this program and circuit

might be useful for experiments on factors that alter the heartbeat rate. What about that last cup of coffee you drank? Can you exert control over your pulse rate with your mind? What is the cause of the statistical fluctuation in heartbeat rates when a person is simply relaxed? What happened to your heartbeat rate when that pretty girl walked by?

**DIGITAL CARDIOTACHOMETER PROGRAM**

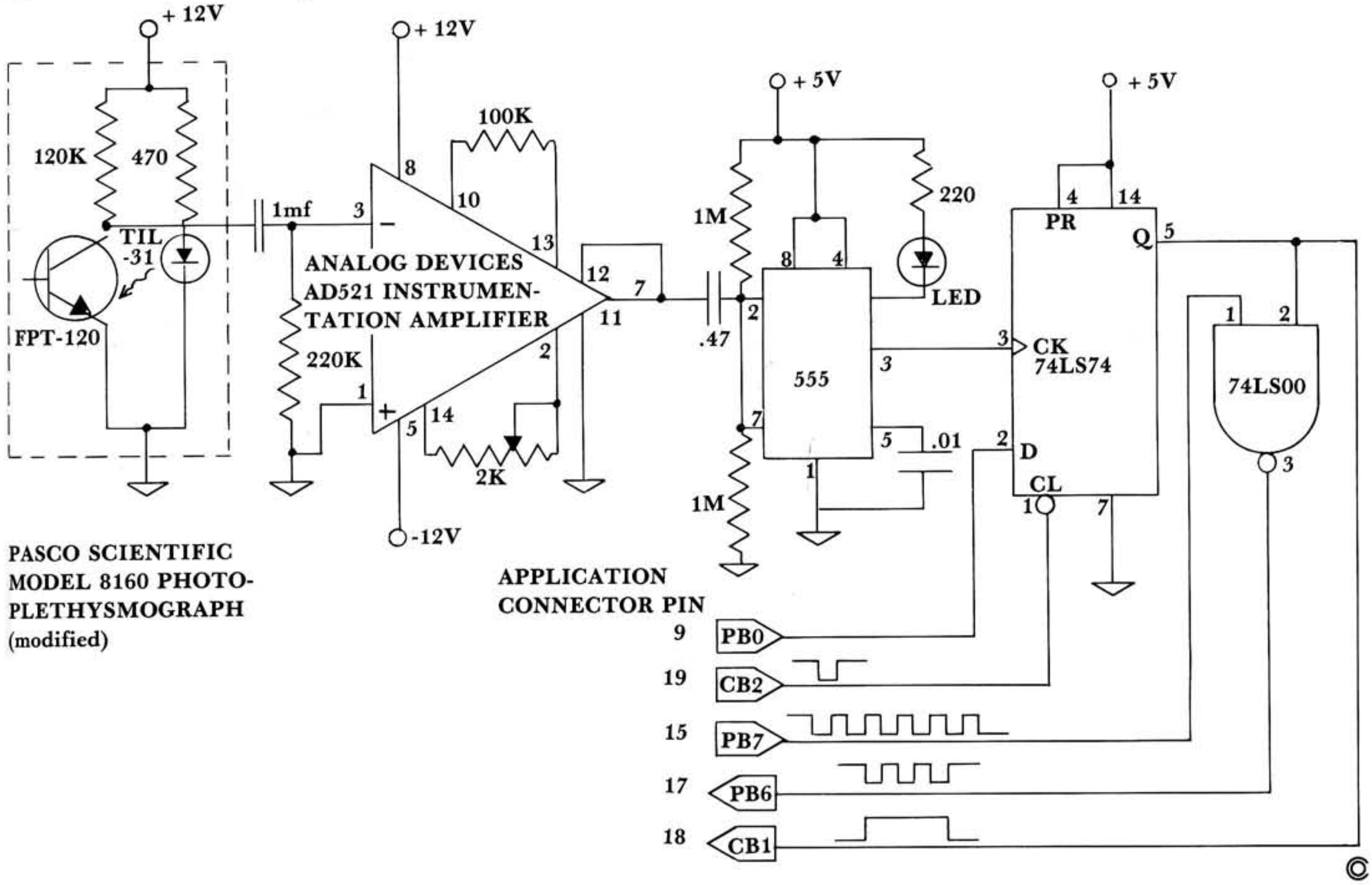
```
10 POKE 04,00; POKE 05,14
20 Y = USR(0)
30 X = 65536*PEEK(51) + 256* PEEK(50) + PEEK(49)
40 R = 600000/X
50 R = INT(R + .5)
60 PRINT R; "PULSES/MIN"
70 GO TO 20
80 END
```

```
0E00 A9 B0     START    LDA $B0      Initialize PCR on the 6522. CB 2 in output
0E02 8D 0C A0           STA PCR      pulse mode. Transition on CB1 sets flag.
0E05 A9 81              LDA $81      Initialize DDRB so PB0 and PB7 are output
0E07 8D 02 A0           STA DDRB     pins.
0E0A 8D 00 A0           STA PBD      Set PB0 and PB7 to logic one.
0E0D CE 00 A0           DEC PBD      Make D-input on flip-flop logic zero.
0E10 A9 E0              LDA $E0      Set up ACR so T1 is in free-running mode
0E12 8D 0B A0           STA ACR      and T2 counts pulses.
0E15 A9 A0              LDA $A0      Set up IER so T2 produces interrupts when
0E17 8D 0E A0           STA IER      it counts through zero.
0E1A A9 30              LDA $30      Set period of pulse train from PB7 to be
0E1C 8D 06 A0           STA T1LL     100 microseconds.
0E1F A9 00              LDA $00
0E21 8D 05 A0           STA T1LH     Start pulse train from PB7.
0E24 A9 00              LDA $00      Clear interrupt counter.
0E26 85 33              STA PLSHI    This location contains number of interrupts.
0E28 A9 FF              LDA $FF      Initialize the T2 counter to count down
0E2A 8D 08 A0           STA T2LL     from $FFFF.
0E2D 8D 09 A0           STA T2CH
0E30 EE 00 A0           INC PBD      Set D-input to logic one. Next pulse from
0E33 58                 CLI          plethysmograph will start timing.
0E34 AD 0D A0  WAIT     LDA IFR      Check flag to see if timing has started.
0E37 29 10              AND $10      Mask all except bit four of the IFR.
0E39 F0 F9              BEQ WAIT     Loop here until a pulse starts the timing.
0E3B A9 00              LDA $00      Clear PCR to prevent clearing the 74LS74.
0E3D 8D 0C A0           STA PCR
0E40 AD 09 A0  LOAF     LDA T2CH     Read the timer. Wait here until about
0E43 C9 F4              CMP $F4      one-half the pulse period has passed
0E45 B0 F9              BCS LOAF     before setting D-input to logic zero
0E47 CE 00 A0           DEC PBD      at the next pulse.
0E4A AD 0D A0  LOITER   LDA IFR      Read the flag register. Has the next
0E4D 29 10              AND $10      pulse occurred?
0E4F F0 F9              BEQ LOITER   No. Then wait here.
0E51 A9 FF              LDA $FF      Yes. Then count the pulses that have
0E53 38                 SEC          occurred (from PB7 to PB6).
0E54 ED 08 A0           SBC T2CL     Low order byte of PB7 pulse count.
0E57 85 31              STA PLSLO
0E59 A9 FF              LDA $FF      Get middle byte of PB7 pulse count.
0E5B ED 09 A0           SBC T2CH
0E5E 85 32              STA PLSMI    PLSHI, PLSMI, and PLSLO are read by the
0E60 78                 SEI          BASIC program.
0E61 4C D1 C0           JMP BASIC    Return to BASIC program.
```

**INTERRUPT ROUTINE: SET IRQ VECTOR TO $0E65**

```
0E65 48        INTRPT   PHA          Save accumulator on the stack.
0E66 A9 FF              LDA $FF      Restart T2 by reloading it.
0E68 8D 09 A0           STA T2CH
0E6B 38                 SEC          Now increment the interrupt counter, PLSHI.
0E6C D8                 CLD
0E6D A9 00              LDA $00
0E6F 65 33              ADC PLSHI
0E71 85 33              STA PLSHI    Result into PLSHI.
0E73 68                 PLA          Recall accumulator contents.
0E74 40                 RTI          Return to the machine language subroutine.
```

Figure 1.
Interface circuit for the digital cardiotachometer.



# Saving Data Matrices With Your SYM-1

George Wells

This article describes a machine-language program that enables BASIC data matrices to be saved on cassette tape and loaded back into the computer at a later time. There have already been several attempts to perform this function, but most of them suffer in one or more of the following ways:

1. They will not allow the BASIC program to be modified.
2. They will not allow the BASIC data to be loaded into a different program.
3. They will not allow selected data to be saved on tape.
4. They will not allow string data to be saved on tape.
5. They are clumsy to use, requiring PEEKing and POKEing.

The program described here overcomes all of these problems. It will run only on a SYM-1 with the new MONITOR 1.1 ROM and will require extensive modification to enable it to run on other machines.

**Functional Description Of Program**

After the machine-language program is in memory and BASIC is running, all that is required to save a matrix is a statement of the form:

**MATRIX (1,2,3) = USR (SAV,ID)**

Where MATRIX (1,2,3) is any kind of matrix (numeric, integer or string) of any size with any number of dimensions, SAV is a previously defined variable pointing to the SAVE.MAT machine-language program and ID is a variable in the range of 0 to 127 which is the tape ID file number. This statement can be entered as a direct command after a program has been run or it can be used anywhere in a program, even in a loop. If you want to save the entire MATRIX, then use the same subscripts that were used to DIMension and the MATRIX. You can save a portion of the MATRIX by making the last subscript a smaller number than its DIMension. SAV and ID cannot be matrix variables.

To load a matrix back into the computer use a similar statement of the form:

**MATRIX (1,2,3) = USR (LOA,ID)**

Where LOA is a previously defined simple variable pointing to the LOAD.MAT machine-language program and the other variables are the same ones used to save the MATRIX.

If you have implemented a second cassette control for your SYM-1 (see MICRO 18:5) then the proper cassette will be turned on for a LOAd or a SAVe operation and you can write programs that in effect handle a very large data base by partitioning it into smaller chunks that are read in from one recorder, operated on and read out to the other recorder automatically.

**Description Of Program Implementation**

**Step 1:** Deposit and Verify the OBJECT LISTING. If you have only 4K of RAM, do this at 0EE6 and then change all of the 1F's to 0F's. This can be done easily with .M 1F,EE6-FFF (CR) followed by 11 sets of 0FG (no (CR)'s). Do another verify (.V EE6-FFF) which should give a checksum of 8747.

**Step 2:** Jump to BASIC (J 0) and use 7910 for the size or 3814 if you have 4K of RAM.

**Step 3:** Enter a program such as:

```
100 SAV = &"1F07" (or SAV = &"0F07" for 4K)
110 POKE 42544,10: REM LONG TAPE DELAY
120 FOR I = 0 TO 10
130 A%(I) = I
140 A(I) = SQR(I)
150 A$(I) = CHR$(I + 65)
160 NEXT I
150 A%(10) = USR(SAV,1)
180 A(10) = USR(SAV,2)
190 A$(10) = USR(SAV,3)
```

**Step 4:** Rewind a tape and start it in record mode.

**Step 5:** RUN the program.

**Step 6:** When BASIC responds with OK, type NEW and enter a second program such as:

```
100 LOA = & "1EE6" (or LOA = &"0EE6"
for 4K)
110 A%(10) = USR(LOA,1)
120 A(10) = USR(LOA,2)
130 A$(10) = USR (LOA,3)
140 FOR I = 0 TO 10
150 PRINT A%(I), A(I), A$(I)
160 NEXT I
```

**Step 7:** Rewind the tape and start it in play mode.

**Step 8:** RUN the program. After all the matrices are read in from tape they will be printed on the terminal.

**Step 9:** In case the computer has trouble reading a file, rewind the tape and restart it in play mode. If you want to abort the tape read process, hold the BREAK key on the terminal down until the tape stops. You can CONTinue from this point if you want to; however, the matrix that couldn't be read in will be cleared to zeroes or nulls.

### Description Of Program Operation

The program stores two copies of each file on tape to provide an automatic back-up in case of error during loading. Between files, the tape delay is set to its minimum possible value (about 1.5 seconds) and set back to its default value (about 6 seconds) at the end of the routine. You can change these values if you have special requirements. Also, you can set the tape delay to a larger value at the beginning of tape operations to automatically move the tape off its leader, as in the first sample program above at line 110.

For numeric and integer matrices, the elements themselves are stored directly on tape with no manipulation since they are already in contiguous order. The start and stop addresses are determined from "foot-prints" on page zero left over from the normal matrix interpretation done by BASIC.

For string matrices the procedure is considerably more complex. About one half of the total code is involved in the special requirements of strings since they are not stored together in one place. The best we can do is rearrange things until all the string information is contained in two separate files which can be stored on tape. The first of these files consists of three bytes per string. The first byte is the length of the string and the last two are a pointer or address to where the ASCII characters making up the string are stored. This first file is already in contiguous order but before it can be used the second file must be created since the first file contains pointers to the second file. The second file is created by going through all the string pointers (in reverse order) and copying each string into unused memory space using two of the BASIC interpreter routines which leave the strings themselves in one continuous block. When the first file is stored it is given an ID greater than 127 (most significant bit is set) so the load routine can distinguish it from the second file.

The routines determine when a string matrix is being operated on by pulling six bytes off the stack and branching on the condition of a zero which indicates a non-string matrix. These six bytes plus one more are

```
>ASSEMBLY LISTING:

                ID          .DE $6F         COPY OF TAPE ID
                MAT.START   .DE $A8         POINTER TO FIRST ELEMENT OF MATRIX
                MAT.STARTC  .DE $70         COPY OF *MAT.START
                MAT.CUR.EL  .DE $99         POINTER TO MATRIX CURRENT ELEMENT
                MAT.ENDC    .DE $6D         COPY OF END OF MATRIX FOR ABORT
                EL.SIZE     .DE $78         NUMBER OF BYTES PER ELEMENT
                CUR.STRING  .DE $BF         CURRENT STRING TO BE TRANSFERRED
                NEW.STR.PN  .DE $83         POINTER TO NEW STRING
                NEW.STRING  .DE $D2A9       GET LOCATION OF STRING OF LENGTH=A
                XFR.STRING  .DE $D42F       TRANSFER STRING TO NEW LOCATION
                ACCESS      .DE $8B86       SYSTEM MONITOR RAM UNPROTECT
                INJISV      .DE $8392       CARRY SET MEANS BREAK
                F1          .DE $8723       MIDDLE OF MONITOR FILL ROUTINE
                DIG         .DE $A400       FIRST DIGIT OF DISPLAY
                P1          .DE $A64E       TAPE ID PARAMETER
                P2          .DE $A64C       TAPE START ADDRESS
                P3          .DE $A64A       TAPE STOP ADDRESS + 1
                TAPDEL      .DE $A630       TAPE DELAY LOCATION
                TAPE.DELAY  .DE 4           TAPE DELAY DEFAULT VALUE
                J.SAVET     .DE $C6         BASIC JUMP VECTOR TO SAVE TAPE
                J.LOADT     .DE $C9         BASIC JUMP VECTOR TO LOAD TAPE
                            .BA $1EE6
                            .OS

1EE6- 20 2E 1F  LOAD.MAT    JSR INIT.PARMS        INITIALIZE TAPE PARAMETERS
1EE9- 68                    PLA                    FOR FIRST FILE
1EEA- 68                    PLA
1EEB- 68                    PLA                   THROW AWAY 7 STACK BYTES
1EEC- 68                    PLA                    6 NOW, 1 LATER
1EED- 68                    PLA
1EEE- 68                    PLA
1EEF- F0 0B                 BEQ LOAD.MAT6         BRANCH IF NOT STRING MATRIX
1EF1- 20 C9 1F              JSR LOADT.HS          GET STRING POINTER FILE
                                                   (ID MUST BE > 127)
1EF4- 20 5F 1F              JSR ORD.STRING        RESERVE STRING FREE SPACE
1EF7- F0 0C                 BEQ LOAD.MAT7         BRANCH IF ALL STRINGS NULL
1EF9- 20 9E 1F              JSR STR.PARMS         SET UP STRING PARAMETERS
1EFC- A5 6F     LOAD.MAT6   LDA *ID               ID MUST BE < 128
1EFE- 29 7F                 AND #$7F
1F00- 85 6F                 STA *ID
1F02- 20 C9 1F              JSR LOADT.HS          GET LAST FILE
1F05- 68        LOAD.MAT7   PLA                   THROW AWAY LAST STACK BYTE
1F06- 60                    RTS                   RETURN TO BASIC

1F07- 20 2E 1F  SAVE.MAT    JSR INIT.PARMS        INITIALIZE TAPE PARAMETERS
1F0A- 68                    PLA                    FOR FIRST FILE
1F0B- 68                    PLA
1F0C- 68                    PLA                   THROW AWAY 7 STACK BYTES
1F0D- 68                    PLA                    6 NOW, 1 LATER
1F0E- 68                    PLA
1F0F- 68                    PLA
1F10- F0 0B                 BEQ SAVE.MAT6         BRANCH IF NOT STRING MATRIX
1F12- 20 5F 1F              JSR ORD.STRING        PUT MATRIX STRINGS IN ORDER
1F15- F0 0D                 BEQ SAVE.MAT7         BRANCH IF ALL STRINGS NULL
1F17- 20 BC 1F              JSR SAVET.2.HS        SAVE 2 COPIES OF POINTERS
                                                   WITH ID > 127
1F1A- 20 9E 1F              JSR STR.PARMS         SET UP STRING PARAMETERS
1F1D- A5 6F     SAVE.MAT6   LDA *ID               MAKE ID < 128
1F1F- 29 7F                 AND #$7F
1F21- 8D 4E A6              STA P1
1F24- 20 BC 1F  SAVE.MAT7   JSR SAVET.2.HS        SAVE 2 COPIES OF LAST FILE
1F27- A9 04                 LDA #TAPE.DELAY       RESTORE TAPE DELAY DEFAULT
1F29- 8D 30 A6              STA TAPDEL
1F2C- 68                    PLA                   THROW AWAY LAST STACK BYTE
1F2D- 60                    RTS

1F2E- 20 86 8B  INIT.PARMS  JSR ACCESS
1F31- 98                    TYA                   PASS ID TO PARM 1 BUT
1F32- 09 80                 ORA #$80               MAKE ID > 127
1F34- 8D 4E A6              STA P1
1F37- 85 6F                 STA *ID               ALSO SAVE COPY FOR STRINGS
1F39- A5 A8                 LDA *MAT.START        PASS MATRIX START TO PARM 2
1F3B- 8D 4C A6              STA P2
1F3E- 85 70                 STA *MAT.STARTC       ALSO SAVE COPY TO ORDER
1F40- A5 A9                 LDA *MAT.START+1       STRINGS
1F42- 8D 4D A6              STA P2+1
1F45- 85 71                 STA *MAT.STARTC+1
1F47- A5 99                 LDA *MAT.CUR.EL       CALCULATE PARM 3
1F49- 85 BF                 STA *CUR.STRING       SAVE COPY TO ORDER STRINGS
1F4B- 18                    CLC
1F4C- 65 78                 ADC *EL.SIZE          ADD ELEMENT SIZE TO INCLUDE
1F4E- 8D 4A A6              STA P3                 CURRENT ELEMENT
1F51- 85 6D                 STA *MAT.ENDC         SAVE COPY FOR ABORT
1F53- A5 9A                 LDA *MAT.CUR.EL+1
1F55- 85 C0                 STA *CUR.STRING+1
1F57- 69 00                 ADC #0                PROPAGATE CARRY
1F59- 8D 4B A6              STA P3+1
1F5C- 85 6E                 STA *MAT.ENDC+1
1F5E- 60                    RTS

1F5F- A9 00     ORD.STRING  LDA #0                CLEAR NON-NULL STRING FLAG
1F61- 85 9A                 STA *MAT.CUR.EL+1
1F63- A0 00     ORD.STR.1   LDY #0
1F65- B1 BF                 LDA (CUR.STRING),Y    LENGTH OF CURRENT STRING
```

normally used to pass pertinent information to the calling routine. However, as they are used here, we don't want the calling routine to store the returned value in the specified matrix, so we simply discard the seven stack bytes. In fact, under normal conditions, it is not possible to specify a string variable in a USR statement, but by pulling seven bytes off the stack, we avoid the type mismatch (TM) error test (which fortunately is done after returning from USR) and return one level deeper which causes the BASIC interpreter to go on to the next statement.

If an attempt is made to store a string matrix that consists entirely of null strings, then only the first file is stored since there will be nothing in the second file. Special tests are made in both the save and load routines to handle this case. Also, it is necessary to use the end of the last non-null string as the end of the second file since to use the pointer of a null string would result in a meaningless tape stop address. Special tests are used to perform this function.

It is a good idea to eliminate DATA statements from a program after they are used to initialize a matrix that is stored on tape since they will only take up memory in future runs. If during the ordering of strings the memory is actually used up, a OM (out of memory) error will occur.

No record is kept on the tape of the name of the matrix or its size and no tests are performed to verify these things. However, if the file does not have the correct number of bytes in it, it will fail the normal tape error tests. The intention is to provide a means for a program or operator to save and retrieve data conveniently, but the task of remembering the size of the matrix or portion of matrix remains with the program or operator. Unlike the usual BASIC command for LOADing programs, if you don't know how big the matrix is that is on a tape, it can be very difficult to load it, so be organized in your use of these routines and they will serve you well.

```
1F67- F0 1C     0950            BEQ ORD.STR.3         BRANCH IF LENGTH = 0 (NULL)
1F69- A6 9A     0960            LDX •MAT.CUR.EL+1
1F6B- D0 08     0970            BNE ORD.STR.2         BRANCH IF NON-NULL STRING
                0980                                   ALREADY FOUND
1F6D- A6 BF     0990            LDX •CUR.STRING       COPY POINTER TO LAST NON-
1F6F- 86 99     1000            STX •MAT.CUR.EL        NULL STRING
1F71- A6 C0     1010            LDX •CUR.STRING+1
1F73- 86 9A     1020            STX •MAT.CUR.EL+1
1F75- 20 A9 D2  1030 ORD.STR.2  JSR NEW.STRING        GET NEW LOCATION FOR STRING
1F78- 20 2F D4  1040            JSR XFR.STRING        TRANSFER TO NEW LOCATION
1F7B- C8        1050            INY                   Y = Y + 1 = 1
1F7C- A5 83     1060            LDA •NEW.STR.PN       COPY NEW STRING POINTER
1F7E- 91 BF     1070            STA (CUR.STRING),Y
1F80- A5 84     1080            LDA •NEW.STR.PN+1
1F82- C8        1090            INY
1F83- 91 BF     1100            STA (CUR.STRING),Y
1F85- A5 BF     1110 ORD.STR.3  LDA •CUR.STRING       GET NEXT STRING POINTER
1F87- 38        1120            SEC                    (WORKING FROM LAST TO
1F88- E5 78     1130            SBC •EL.SIZE           FIRST SO SUBTRACT)
1F8A- 85 BF     1140            STA •CUR.STRING
1F8C- A6 C0     1150            LDX •CUR.STRING+1
1F8E- B0 01     1160            BCS ORD.STR.4
1F90- CA        1170            DEX                   PROPAGATE BORROW
1F91- 86 C0     1180 ORD.STR.4  STX •CUR.STRING+1
1F93- E4 71     1190            CPX •MAT.STARTC+1     TEST FOR ALL STRINGS DONE
1F95- D0 CC     1200            BNE ORD.STR.1
1F97- C5 70     1210            CMP •MAT.STARTC
1F99- B0 C8     1220            BCS ORD.STR.1
1F9B- A5 9A     1230            LDA •MAT.CUR.EL+1     SET Z IF ALL STRINGS NULL
1F9D- 60        1240            RTS
                1250
1F9E- A5 83     1260 STR.PARMS  LDA •NEW.STR.PN       START OF STRINGS TO PARM 2
1FA0- 8D 4C A6  1270            STA P2
1FA3- A5 84     1280            LDA •NEW.STR.PN+1
1FA5- 8D 4D A6  1290            STA P2+1
1FA8- A0 00     1300            LDY #0
1FAA- B1 99     1310            LDA (MAT.CUR.EL),Y    END OF STRINGS TO PARM 3
1FAC- 18        1320            CLC
1FAD- C8        1330            INY
1FAE- 71 99     1340            ADC (MAT.CUR.EL),Y    ADD LENGTH OF LAST NON-NULL
1FB0- 8D 4A A6  1350            STA P3                 STRING TO ITS POINTER TO
1FB3- C8        1360            INY                    END OF STRINGS
1FB4- B1 99     1370            LDA (MAT.CUR.EL),Y
1FB6- 69 00     1380            ADC #0
1FB8- 8D 4B A6  1390            STA P3+1
1FBB- 60        1400            RTS
                1410
1FBC- 20 C4 1F  1420 SAVET.2.HS JSR SAVET.HS          SAVE 2 COPIES IN HI-SPEED
1FBF- A9 01     1430            LDA #1                 MODE WITH MINIMUM DELAY
1FC1- 8D 30 A6  1440            STA TAPDEL             BETWEEN THEM
1FC4- A0 80     1450 SAVET.HS   LDY #$80
1FC6- 4C C6 00  1460            JMP J.SAVET           JUMP THROUGH BASIC VECTOR
                1470
1FC9- 20 92 93  1480 LOADT.HS   JSR INJISV            TEST FOR BREAK
1FCC- B0 14     1490            BCS ABORT
1FCE- A9 FF     1500            LDA #$FF              GET ANY FILE FROM TAPE
1FD0- 8D 4E A6  1510            STA P1
1FD3- A0 80     1520            LDY #$80
1FD5- 20 C9 00  1530            JSR J.LOADT           JUMP THROUGH BASIC VECTOR
1FD8- B0 EF     1540            BCS LOADT.HS          REPEAT IF BAD LOAD
1FDA- AD 00 A4  1550            LDA DIG               GET ID
1FDD- C5 6F     1560            CMP •ID
1FDF- D0 E8     1570            BNE LOADT.HS          REPEAT IF WRONG ID
1FE1- 60        1580            RTS
                1590
1FE2- 68        1600 ABORT      PLA                   DISCARD EXTRA STACK BYTES
1FE3- 68        1610            PLA
1FE4- 68        1620            PLA
1FE5- A9 00     1630            LDA #0                CLEAR ABORTED MATRIX
1FE7- A6 70     1640            LDX •MAT.STARTC       SET UP FOR MONITOR FILL
1FE9- A4 71     1650            LDY •MAT.STARTC+1
1FEB- 86 FE     1660            STX •$FE
1FED- 84 FF     1670            STY •$FF
1FEF- A4 6E     1680            LDY •MAT.ENDC+1
1FF1- A6 6D     1690            LDX •MAT.ENDC
1FF3- D0 01     1700            BNE ABORT.4
1FF5- 88        1710            DEY                   SUBTRACT ONE FROM END
1FF6- CA        1720 ABORT.4    DEX
1FF7- 8E 4A A6  1730            STX P3
1FFA- 8C 4B A6  1740            STY P3+1
1FFD- 4C 23 87  1750            JMP F1                FILL AND RETURN
                1760            .EN
```

©

# Part 1 of a series: OSI ROMs

T. R. Berger

There seems to be some curiosity about OSI's non-BASIC ROMs, i.e. ones above address $F800. The schematic for the C1, 2, 4, and 8 shows a 2K ROM (type 2316B). Disassembly shows that the C1 has the full 8 pages of this ROM addressed in memory locations $F800 to $FF00, but the C2, 4, and 8 have only 3 pages of this 8 page ROM appearing in memory. Actually, there is only one 2K ROM used in all these machines! (The old C2's without polled keyboard and the C3 with serial monitor or hard disk are exceptions which we will ignore.)

The C2, 4, and 8 have special address selecting circuitry which allows the computer to choose and address any 3 of the 8 pages in this 2K ROM. The C1 does not contain this special circuitry (presumably to cut costs) so that all 8 pages of the ROM appear in memory whether or not they are needed. The C1 uses 4 of the 8 pages and the remaining 4 pages fall where they will, misaddressed and unrunable. Since the C1 is the only machine with all 8 pages, let us use C1 addresses to describe the various pages of this ROM.

The page in which a segment of code appears in the C1 is not necessarily the page in which the code is written to run. For example, Cold Start for ROM BASIC C2, 4, and 8 computers was written to run in page $FF but appears in page $FB in the C1. In particular, in the C1 at $FB43 we see JSR $FFB8. The requested subroutine now appears at address $FBB8 in the C1 and $FFB8 is actually the third byte of a jump instruction. If the computer is asked to run this code, this subroutine jump will send the computer to limbo. Thus the code (with minor exceptions of no importance) in page $FB of the C1 will not run.

Table 1 gives a summary of the various functions of this ROM. The first column gives the C1 page number, the second column gives the model numbers of Challengers which use the given page of code (e.g. 4 means C4); the third column gives the type of machine (i.e. ROM BASIC or Disk); the fourth column gives the function of the page; and finally, the fifth column tells in which page the code was written to run.

C1 owners with a disassembler may read the code which all of us run. The rest of us will have to make do with the pages we actually use. However, not much is missed since the code is highly redundant. Commercial computer users know that the most costly facet of computing is software development. OSI can sell its computers at low cost because of a policy minimizing software development costs. That is, if a program is written for one machine, and by patching existing code it will run on another (even though at less than peak efficiency) then, by all means, patch. Thus the C1 code in these ROMs is just patched versions of the code written for the C2, 4, and 8. If this ROM were rewritten carefully, it would easily fit into 1K and run on all machines. But the larger ROMs come much more cheaply than rewriting the code. MORAL: Hardware is cheap, software is expensive. Businessmen know this; we do too when we yell 'software ripoff!'

Since C1 run pages differ only in patches and relocation (with precious little relocation) from C2, 4, and 8 run pages, it is necessary to only understand about half of this ROM in order to understand it all. The fundamental pages are listed in Table 2. When the code was patched to run on the C1, every effort was made not to move addresses. For example, where the machine monitor for the C2, 4, and 8 resets a P1A not available on the C1, (C1 addresses $FA04 to $FA0B), the corresponding code for the C1 is filled with NOP instructions (C1 addresses $FE04 to $FE0B).

I am writing a series of articles on the OS65D operating system. The first article, on the Kernel, should appear in the next issue. At least one later article in this series will be devoted to this ROM. In particular, much more detail on these ROMs will appear in the near future. If you have urgent questions, you may send a stamped self-addressed envelope to me at:

Tom Berger
10670 Hollywood Blvd.
Coon Rapids, MN. 55433

**TABLE 1**

| C1P PAGE | MACHINE | TYPE | FUNCTION | RUN PAGE |
|---|---|---|---|---|
| $F8 | 2,4,8 | DISK | BOOT | $FF |
| $F9 | 2,4,8 | ALL | KEYBOARD | $FD |
| $FA | 2,4,8 | ALL | MONITOR | $FE |
| $FB | 2,4,8 | ROM BASIC | BOOT | $FF |
| $FC | 1 | DISK | BOOT | $FC |
| $FD | 1 | ALL | KEYBOARD | $FD |
| $FE | 1 | ALL | MONITOR | $FE |
| $FF | 1 | ALL | BOOT | $FF |

©

# ALL ABOUT OSI MICROSOFT BASIC-IN-ROM, VERSION 1.0 REV 3.2

by Edward H. Carlson
**Published by the author**
**3872 Raleigh Drive**
**Okemos, MI, 48864**
**8½ x 11 inches Soft-cover, 60 Pages, $8.95**
Review by Charles L. Stanford

This book would almost certainly have saved me from many very frustrating hours of pouring through OSI's so-called "Manuals" and several of the general texts on BASIC during the first months I owned my C1P. Would you believe, I discovered how to use the immediate mode by accident, in about the fifth week? Mr. Carlson covers the subject very nicely on the first page of his new book.

Granted, BASIC is BASIC, to a very large degree. It's not too hard to convert from one computer's BASIC to another's. Most commands and functions act alike. But there are significant differences in some areas; I can never remember the exact result of some of the functions I seldom use, such as TAB and INPUT. So every encounter with those commands in a published program requires some searching in the files. This reference book ends that.

Mr. Carlson states in the introduction that he is presenting OSI BASIC on two levels; pure BASIC, and then the underlying principles of program code storage, pointers, flags, and the way programs really work. He succeeds admirably in the first, with only minor exceptions. Each of the commands, statements, functions, and operators is listed and discussed in detail. Many examples and suggestions are included, and useful combinations of functions are presented. In particular, I learned some things about the USR function I could only guess at before (and I use it heavily). A very few of the functions covered could have stood a heavier treatment. WAIT I,J,K and the Boolean Operators AND, OR, and NOT are, as is usual, glossed over, apparently under the theory that everyone understands Boolean Algebra. It is very elementary, but many hobbyists haven't yet seen its usefulness, and may never until more of its advantages are more thoroughly pointed out and explored. Be assured, though, that these are minor faults - the coverage of BASIC is very thorough. This manual, together with a more generalized text (such as Basic BASIC) should meet any programmer's needs.

Mr. Carlson's treatment of his second task is not quite so successful, although still very thought-provoking and useful. He has caused some considerable loss of sleep, while I explore the machine codes his lists and tables point out. There is an extremely well annotated memory map for pages $00 through $03, as well as a listing of the Monitor ROM from $FE00 to $FFFF. The ROM listing is for the C2-4P, but differences in the C1P ROM are called out in a separate list. I think most OSI owners have figured out by now that the BASIC ROMs are identical for all models, with only the Monitor ROMs being different.

Other useful discussions are presented on the format and programming of both BASIC and Machine Language tapes, floating point numbers, and the way the program stores variables. Less successful sections cover the source code and two's complement binary numbers. The only section that fails badly, however, is the one on OSI's Big Bug. I refer, of course, to the String Array Garbage Collection glitch. The author presents what might appear to be a really simple solution. It does, in fact, work. But it's virtually impossible to overlay on an existing program which uses a lot of concatenated string arrays. There are more effective solutions to this bug.

All in all, I think that any of several sections of this very well presented manual are worth the purchase price. I recommend it to the hobbyist and to the serious programmer alike. ©

# Fast Graphics On The OSI C1P

Charles L. Stanford

I am sure that almost without exception, OSI C1P and Superboard II owners get a major part of their computer fun from games. The relative ease of graphics programming using the game symbols in the Character Generator ROM allows even the novice BASIC programmer a lot of freedom for creating exciting and fastmoving figures. However, even OSI's very fast BASIC still leaves a lot to be desired when more than a few dozen screen locations are changed, resulting in distracting image changes and in slow execution of commands.

This article will present a "cookbook" version of a machine language graphics subroutine. The BASIC user with little or no background in Machine or Assembly language programming will be able to use the material presented here to "plug in" his own graphics. The more advanced programmer will be able to adapt the concepts shown to complicated, interactive graphics games and simulations.

**The use of graphics in games**

While many exciting and worthwhile games are played without graphics (such as most versions of Starwars, Star Trek, Quest, etc.), most family fun games revolve around the manipulation of graphics figures or objects on the screen. Six Gun Shootout, Tank, Lunar Lander, and many others are "made" by their graphics interactions. But a game like Six Gun written solely in BASIC is slowed down so badly by the time required to POKE two gunfighters and a few cacti that it looses a lot of its **Oomph!** Even a near-instantaneous screen routine (instead of scrolling) only helps a little when 50 or 100 or more POKE's are needed every time a move is made.

The method presented in this article does have a few limitations, mainly in the opportunity for typing errors when entering the many DATA statements, and in the need to carefully plan and structure each and every character before starting to enter the program. But the results are really worth it. Every program I've written since devising this method has had to be slowed down with time delay loops, which is sure better than the alternative.

**The USR Function**

The OSI Graphics Reference Manual and the BASIC in ROM Manual are ridiculously brief in describing the use of the USR function. On the other hand, there isn't that much knowledge needed for an elementary application such as this one. In general, you will need to be capable of converting Hex numbers to Decimal quickly and accurately; the only other requirement is to be able to write a fairly structured BASIC program.

When the USR function is called by a line such as 10 X = USR (X), the program goes to the machine language program subroutine pointed to by the data at memory locations $000A and $000B (Decimal #11 and #12). The 6502 processor needs a 16 bit address to jump to a subroutine. But each memory location only has 8 bits. So the processor reads the eight lower (right-most) bits of the address from the lower of two adjacent memory addresses, and the eight higher (left-most) bits from the higher. Thus, a machine language routine at address $0222 would be called if memory locations $000A and $000B held $22 and $02 respectively. These numbers would be inserted in these two RAM locations by a line in BASIC such as 20 POKE 11,34 : POKE 12,2. Note that the Hex numbers have to be converted to their Decimal equivalents for BASIC. As you can see, many machine programs starting at as many different memory locations could be called, merely by changing the data at $000A and $000B before calling the USR function. I'll describe how to write and insert the actual machine language subroutines a bit later in this article.

After a machine language subroutine is called, it acts very similarly to a BASIC program, in that each instruction is executed in order until a RETURN is encountered. It then returns to the next instruction after the JUMP. The RETURN in machine language is the RTS instruction, which is Hex $60 (or Dec #96).

The advantage of all this is, of course, based in the fact that machine language is somewhere around 1000 times faster than BASIC for most equivalent functions. This feature is particularly useful for graphics programming.

**Machine language programming**

Writing programs in machine language, especially without the use of an Assembler, is a bit tedious. But the end result is often worth the effort. In this case, you, the reader, will use the program itself exactly as presented here, only changing a table which will hold whichever graphics figures needed for your particular game.

The action of the subroutine is quite simple. After a Screen Clear (note that the first eight lines of Listing 1 are as shown in my previous article in Compute II Number 1), a series of graphics characters and their locations relative to a fixed point are read from memory and written to the TV screen.

The routine makes use of one of the 6502 processor's more efficient addressing modes, Zero Page. Line 023A uses the instruction LDA-0,Y, which loads the accumulator with the data residing at a memory location equal to the value of the address pointed to by FE (lo byte) and FF (hi byte) plus the contents of the Y register. These two memory locations at the top of Zero page were chosen as they are easy to

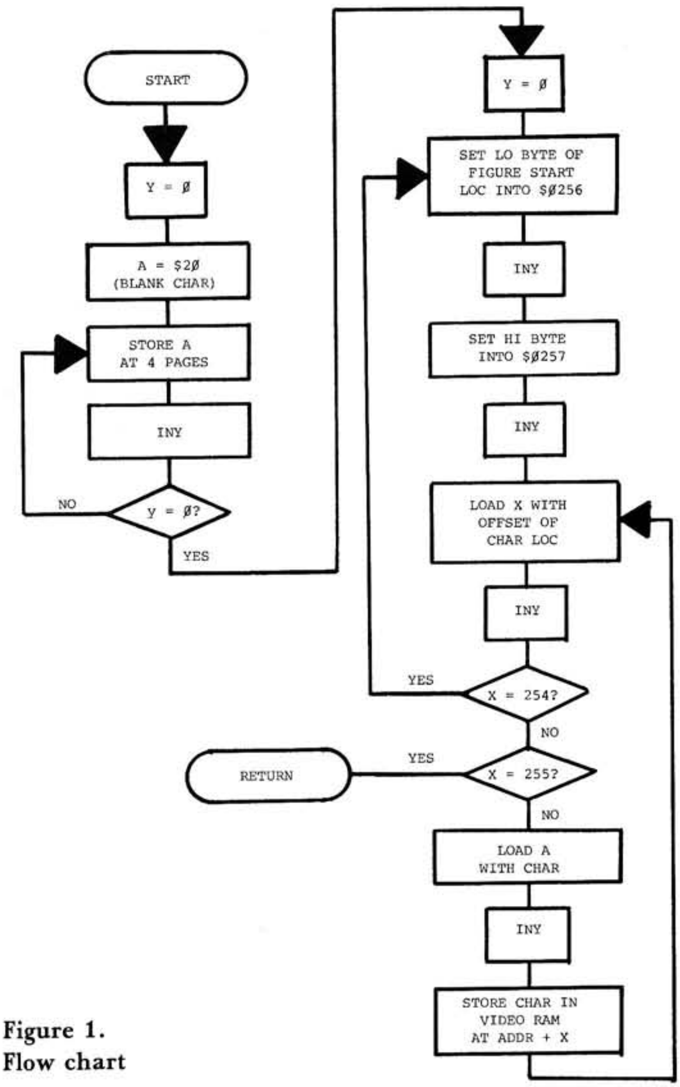



**Figure 1.**
**Flow chart**

remember, and are not used for any other purpose by BASIC or the Monitor. Since the Y register is an eight-bit register, only screen locations between the base address and the base address plus 255 can be called.

Looking at the flow chart (Figure 1), you can see that the first data loaded is the lo byte, then the hi byte, of the screen location. This information is in the table for this particular game, and is deposited within the program to give a start location on the screen for the first graphics figure. Next, the offset of the first character is loaded into the X register, and a check is made to see if the figure is ended. I have selected $FE (254) as a signal that part of the figure is completed, and $FF (255) for the end of the routine. If this data is neither, a graphics symbol will be loaded into the accumulator. This symbol is then deposited into the video refresh memory at the location previously loaded plus the offset in the X register.

This may all sound a bit complicated, but that doesn't make any difference to you, the programmer. The program will take care of itself if the proper data is presented from the symbol table in the correct sequence.

### Developing the screen image

The screen image is developed by following five easy steps, starting with a copy of the C1P Screen Grid. Being a railroad fan, I chose to enact the "Great Train Collision" as a demonstration program. As you can see from Figure 2a, the OSI graphics symbols allow for a very reasonable steam locomotive with tender. After you have drawn whatever figures you need for your game to your satisfaction, enter the screen offsets of all locations, starting at a fixed point at the upper left corner of each figure, as shown in Figure 2b. Finally, write in the character generator code as shown in Figure 2c. The last step is to enter the data into a table as shown in Listing 2. Note that each figure is preceded by a screen address (lo byte first), and ends in either FE or FF. The last figure in the table *must* be FF (255) which causes the subroutine to return to BASIC.

### Converting the Machine Language to BASIC

There are several methods of entering machine or assembly language programs through the use of BASIC programs, including direct entry through the monitor. There are also several ways to save combination BASIC-machine language programs to tape (see Daniel Schwartz' article in Compute II issue 1). However, I prefer the somewhat tedious but easy to debug and modify DATA-POKE method. This is hard to enter error-free, and causes a significant delay on program initiation, but it has the advantage of being more readily understandable, and allows your routines to be changed easily.

You can see that lines 100 through 145 in Listing 3 enter the machine language subroutine into memory, while lines 148 through 199 are the table of offsets and characters for the figures. This is to allow the use of this sequence with other graphics figures of your own choosing. Note also that the subroutine is inserted into RAM at $0222 (#546) in page 2. The page 2 memory from $0222 to $02FA is not used by BASIC, and is thus a good free place for this use. However, make sure your figures don't extend above $02FA. If so, you'll have to locate the table in the top of RAM, necessitating memory protection at Cold Start. The subroutine can still reside at $0222.

### Animating the Figures

Once the graphics subroutine and the figure table are in RAM, the animation routines can be written in BASIC. The addresses needed to animate the demonstration figures are as follows:

| HEX | DEC | DESCRIPTION |
|---|---|---|
| 000A | 11 | Pointer to the starting address |
| 000B | 12 | of the USR subroutine. |
| 00FE | 254 | Pointer to the starting address of |
| 00FF | 255 | the first graphics figure to be called. |
| 0222 | 546 | Actual starting address of the USR subroutine, including screen clear. |
| 0238 | 568 | Actual starting address of the USR subroutine without screen clear. |
| 0260 | 608 | Actual starting address of the first |

| | | | | | |
|---|---|---|---|---|---|
| $Ø222 | AØ ØØ | 546 | 16Ø Ø | LDY-IMM | $ØØ |
| Ø224 | A9 2Ø | 548 | 169 32 | LDA-IMM | $2Ø |
| Ø226 | 99 ØØ D3 | 55Ø | 153 Ø 211 | STA-Y | |
| Ø229 | 99 ØØ D2 | 553 | 153 Ø 21Ø | STA-Y | |
| Ø22C | 99 ØØ D1 | 556 | 153 Ø 2Ø9 | STA-Y | |
| Ø22F | 99 ØØ DØ | 559 | 153 Ø 2Ø8 | STA-Y | |
| Ø232 | C8 | 562 | 2ØØ | INY | |
| Ø233 | DØ F1 | 563 | 2Ø8 241 | BNE | $Ø226 |
| Ø235 | EA EA EA | 565 | 234 234 234 | NOP NOP NOP | |
| Ø238 | AØ ØØ | 568 | 16Ø Ø | LDY-IMM | $ØØ |
| Ø23A | B1 FE | 57Ø | 177 254 | LDA-Ø,Y | LDA Lo Byte Scr Loc |
| Ø23C | 8D 56 Ø2 | 572 | 141 86 2 | STA-ABS | |
| Ø23F | C8 | 575 | 2ØØ | INY | |
| Ø24Ø | B1 FE | 576 | 177 254 | LDA-Ø,Y | LDA Hi Byte Scr Loc |
| Ø242 | 8D 57 Ø2 | 578 | 141 87 2 | STA-ABS | |
| Ø245 | C8 | 581 | 2ØØ | INY | |
| Ø246 | B1 FE | 582 | 177 254 | LDA-Ø,Y | LDA w/ Char Offset |
| Ø248 | AA | 584 | 17Ø | TAX | A to X Register |
| Ø249 | C8 | 585 | 2ØØ | INY | |
| Ø24A | EØ FE | 586 | 224 254 | CPX-IMM | X=254?End Figure |
| Ø24C | FØ EC | 588 | 24Ø 236 | BEQ | Branch to Ø23A |
| Ø24E | EØ FF | 59Ø | 224 255 | CPX-IMM | X=255?End Routine |
| Ø25Ø | FØ Ø8 | 592 | 24Ø 8 | BEQ | Branch to Ø25A |
| Ø252 | B1 FE | 594 | 177 254 | LDA-Ø,Y | Load Character |
| Ø254 | C8 | 596 | 2ØØ | INY | |
| Ø255 | 9D 44 D1 | 597 | 157 68 2Ø9 | STA-ABS,X | Char. to Screen |
| Ø258 | DØ EC | 6ØØ | 2Ø8 236 | BNE | Get another Character |
| Ø25A | 6Ø EA EA | 6Ø2 | 96 234 234 | RTS NOP NOP | End Routine, Return |
| Ø25D | EA EA EA | 6Ø5 | 234 234 234 | NOP NOP NOP | |

**Listing 1.** Machine Language Subroutine

| | | |
|---|---|---|
| **0261** | **609** | graphics figure. These locations hold the video RAM reference address, Lo byte first, for the figure. |
| **029D** | **669** | Actual starting address of the second |
| **029E** | **670** | graphics figure. Also hold the video RAM starting address for this figure. |

It is important to remember that the last two locations will vary depending on the location in RAM chosen to store the table, and according to the length of the figures in the table. The first four sets of addresses will remain the same for all programs.

In the demonstration program, memory location #12 always holds a #2, but the data in location #11 is changed back and forth between #34 and #56; this alternately inserts and omits the screen clear routine at the beginning of the machine language subroutine. If your program only has one figure, or if you end all figures but the last with #254, location #11 would stay at #34. Either method works equally well; chose the alternative which provides easiest animation in BASIC.

Note that if each figure ends with a #254, the data at memory location #254 will not change. But ending the first or any intermediate figure with #255 terminates the subroutine (see flow chart), and the only way to reach the next or subsequent figures is to change the pointer at #254 and #255.

As previously mentioned, the first two numbers in the table at the beginning of each figure are the video RAM address of the upper left hand corner of the figure. Since the video RAM of the C1P contains four pages of 256 bytes each, starting at $D000, the first number in the table at the start of each figure can vary from 0 to 253 (remember, 254 and 255 are signals), and the second can be $D0 through $D3 (#208 through #211). By POKEing different numbers into these locations, the figures can be made to move around the screen. It's better to use the actual first location of each figure in the original DATA statements, to avoid a jerk at the start of the program. After that, any desired location can be POKE'd into these addresses.

In lines 25 and 35 of Listing 2, variables A and B are established as the low bytes of the screen locations for the two figures. Then, in the subroutine starting at line 50, they are POKE'd to 608 and 669 respectively, and then incremented and decremented and POKE'd again to move the characters across the screen toward each other. To make the demonstration program more realistic, the engines are moved across the screen on different lines, and then reappear on a collision course. This is done by line 35, which changes A and B and also changes the hi byte of the right locomotive to #209 from 210, moving it higher on the screen. Variable C determines how far across the screen each figure will move.

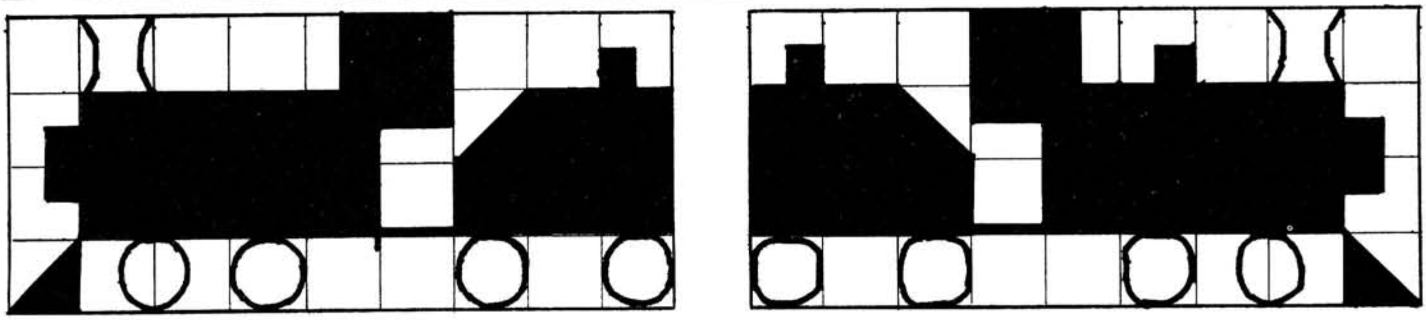
Finally, the routines at lines 200 to 399 give a rough representation of an explosion at the point of collision. I leave to the reader the exercise of adding this to the machine language table. It's not that hard!

**Summary**

In order, the steps for creating your own program using fast machine language graphics are as follows:

**1.** Draw a representation of your selected characters, using the characters in the Graphics Reference Manual as elements.

**2.** In an equivalent number of spaces, enter the offsets of the screen address, starting at a point above and to the left of the figures. If the number of the offset exceeds 253, just split the figure into two or more parts and treat each as a separate entity. They can be



2a

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| * | 1 | | 3 | 4 | 5 | | | 8 |
| 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 4Ø |
| 64 | 65 | 66 | 67 | 68 | 69 | 7Ø | 71 | 72 |
| 96 | 97 | 98 | 99 | | | 1Ø2 | | 1Ø4 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Ø | | | 3 | 4 | 5 | | 7 | |
| 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 4Ø |
| 64 | 65 | 66 | 67 | 68 | 69 | 7Ø | 71 | 72 |
| 96 | | 98 | | | 1Ø1 | 1Ø2 | 1Ø3 | 1Ø4 |

2b

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| * | 2 | | 167 | 157 | 161 | | | 167 |
| 165 | 161 | 161 | 161 | 161 | 155 | 176 | 161 | 161 |
| 166 | 161 | 161 | 161 | 161 | 128 | 161 | 161 | 161 |
| 176 | 224 | 225 | 226 | | | 226 | | 226 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 165 | | | 161 | 156 | 165 | | 2 | |
| 161 | 161 | 178 | 155 | 161 | 161 | 161 | 161 | 167 |
| 161 | 161 | 161 | 128 | 161 | 161 | 161 | 161 | 168 |
| 226 | | 226 | | | 226 | 224 | 225 | 178 |

2c

**Figure 2.** Graphics development

recombined in the BASIC program.
**3.** Likewise, enter the graphics character codes in the equivalent spaces on the grid.
**4.** Create a table which starts with the screen location of the figure (lo byte first, then hi byte); contains, alternately, the offset and code of each character; and ends in #254 or 255. If the figure ends in 254, the subroutine will continue with the next figure in the table. A #255 terminates the subroutine and returns to BASIC.
**5.** Note the starting addresses of each figure, for later use in creating the animation in BASIC.
**6.** Convert both the subroutine and the table into DATA statements. Note that if the table goes past memory location #608 + 144 (#752), it will be necessary to move all or part of it to another location in RAM, such as top of memory.
**7.** Finally, enter the DATA statements and their POKE loops into an appropriate location in your BASIC program, in a manner similar to lines 100-199 of Listing 2. Then procede with animation as in lines 25-99.
**8.** Debugging hints: always save to tape as you go (a program crash is more likely in machine language, and all that tough typing will be lost); insert a BREAK after DATA loops, then use the Monitor to verify the machine language program entry, by single stepping starting at address $0222; insert timing loops at lines 22, 27, 32, and 37 to slow down action if there is a Bug in BASIC.

**Some additional notes**

Another interesting program I've written using this method is Six Gun Shootout. It has two gunfighters (one facing the other) and three cacti. After each shot, the cacti change locations at random. The program ran very slowly when written solely with BASIC POKE's to the screen, but is as fast as you would ever want with machine language graphics. I'll cover this program in a later article, together with instructions for attaching simple up-down-shoot joysticks to save wear and tear on the keyboard.

**Listing 2.** Graphics Table

| | | | | | |
|---|---|---|---|---|---|
| Ø26Ø | 9B | D1 | 6Ø8 | 155 | 2Ø9 |
| Ø262 | Ø1 | Ø2 | 61Ø | 1 | 2 |
| Ø264 | Ø3 | A7 | 612 | 3 | 167 |
| Ø266 | Ø4 | 9D | 614 | 4 | 157 |
| Ø268 | Ø5 | A1 | 616 | 5 | 161 |
| Ø26A | Ø8 | A7 | 618 | 8 | 167 |
| Ø26C | 2Ø | A5 | 62Ø | 32 | 165 |
| Ø26E | 21 | A1 | 622 | 33 | 161 |
| Ø27Ø | 22 | A1 | 624 | 34 | 161 |
| Ø272 | 23 | A1 | 626 | 35 | 161 |
| Ø274 | 24 | A1 | 628 | 36 | 161 |
| Ø276 | 25 | 9B | 63Ø | 37 | 155 |
| Ø278 | 26 | BØ | 632 | 38 | 176 |
| Ø27A | 27 | A1 | 634 | 39 | 161 |
| Ø27C | 28 | A1 | 636 | 4Ø | 161 |
| Ø27E | 4Ø | A6 | 638 | 64 | 166 |
| Ø28Ø | 41 | A1 | 64Ø | 65 | 161 |
| Ø282 | 42 | A1 | 642 | 66 | 161 |
| Ø284 | 43 | A1 | 644 | 67 | 161 |
| Ø286 | 44 | A1 | 646 | 68 | 161 |
| Ø288 | 45 | 8Ø | 648 | 69 | 128 |

| | | | | | |
|---|---|---|---|---|---|
| Ø28A | 46 | A1 | 65Ø | 7Ø | 161 |
| Ø28C | 47 | A1 | 652 | 71 | 161 |
| Ø28E | 48 | A1 | 654 | 72 | 161 |
| Ø29Ø | 6Ø | BØ | 656 | 96 | 176 |
| Ø292 | 61 | EØ | 658 | 97 | 224 |
| Ø294 | 62 | E2 | 66Ø | 98 | 225 |
| Ø296 | 63 | E2 | 662 | 99 | 226 |
| Ø298 | 66 | E2 | 664 | 1Ø2 | 226 |
| Ø9A | 68 | E2 | 666 | 1Ø4 | 226 |
| Ø29C | FF | | 668 | 255 | |
| Ø29D | 83 | D1 | 669 | 131 | 2Ø9 |
| Ø29F | ØØ | A5 | 671 | Ø | 165 |
| Ø2A1 | Ø3 | A1 | 673 | 3 | 161 |
| Ø2A3 | Ø4 | 9C | 675 | 4 | 156 |
| Ø2A5 | Ø5 | A7 | 677 | 5 | 165 |
| Ø2A7 | Ø7 | Ø2 | 679 | 7 | 2 |
| Ø2A9 | 2Ø | A1 | 681 | 32 | 161 |
| Ø2AB | 21 | A1 | 683 | 33 | 161 |
| Ø2AD | 22 | B2 | 685 | 34 | 178 |
| Ø2AF | 23 | 9B | 687 | 35 | 155 |
| Ø2B1 | 24 | A1 | 689 | 36 | 161 |

| | | | | | |
|---|---|---|---|---|---|
| Ø2B3 | 25 | A1 | 691 | 37 | 161 |
| Ø2B5 | 26 | A1 | 693 | 38 | 161 |
| Ø2B7 | 27 | A1 | 695 | 39 | 161 |
| Ø2B9 | 28 | A7 | 697 | 4Ø | 167 |
| Ø2BB | 4Ø | A1 | 699 | 64 | 161 |
| Ø2BD | 41 | A1 | 7Ø1 | 65 | 161 |
| Ø2BF | 42 | A1 | 7Ø3 | 66 | 161 |
| Ø2C1 | 43 | 8Ø | 7Ø5 | 67 | 128 |
| Ø2C3 | 44 | A1 | 7Ø7 | 68 | 161 |
| Ø2C5 | 45 | A1 | 7Ø9 | 69 | 161 |
| Ø2C7 | 46 | A1 | 711 | 7Ø | 161 |
| Ø2C9 | 47 | A1 | 713 | 71 | 161 |
| Ø2CB | 48 | A8 | 715 | 72 | 168 |
| Ø2CD | 6Ø | E2 | 717 | 96 | 226 |
| Ø2CF | 62 | E2 | 719 | 98 | 226 |
| Ø2D1 | 65 | E2 | 721 | 1Ø1 | 226 |
| Ø2D3 | 66 | EØ | 723 | 1Ø2 | 224 |
| Ø2D5 | 67 | E1 | 725 | 1Ø3 | 225 |
| Ø2D7 | 68 | B2 | 727 | 1Ø4 | 178 |
| Ø2D9 | FF | | 729 | 255 | |

©

# Good News For OSI Readers

As you may have read by now, the Editor's Notes in this issue announce the merger of COMPUTE and compute II into one, high quality, monthly magazine.

Given the range of OSI products, we've decided to move you out of the Single-Board Computer Gazette, in the new **COMPUTE!,** and establish an **OSI Gazette.**

We expect to be assisted in this endeavor by your continued input. Articles and programming notes should be addressed to:

**The Editor, COMPUTE!,**
**P.O. Box 5406,**
**Greensboro, NC 27403**
**ATTN: OSI Gazette**

Given your input, we'll have a healthy, OSI Gazette.
Robert Lock

# Modification and Relocation of FOCAL 65-E Into Erasible Prom

William C. Clements, Jr.
Dept. of Chemical & Metallurgical Engineering
The University of Alabama
P. O. Box 2662
University, Alabama 35486

After using FOCAL for awhile, I became interested in storing the machine code in EPROM. Not only would this eliminate much of the waiting for tapes to load, but more important, it would free over 5K of user RAM for other purposes such as storing more FOCAL source code and variables, or for graphics routines.

The relocation of FOCAL and execution of it from PROM is not as straightforward as for some other programs, because the machine code is self-modifying in several places. Also, there are multitudes of data bytes used for address pointers scattered through the program, and these are in such a form that the ordinary kind of relocation routine would not convert them. Thanks to the excellent documentation supplied with FOCAL, I was successful in relocating it in a "clean", non self-modifying form. The code, together with an initialization routine that sets up page zero and other RAM locations used for user statements and to make the code "clean", fits neatly into three 2716's with plenty of room left over for other modifications such as tape load and save, "user" function, etc., which I have added to my version of FOCAL as well.[1] The modifications which follow are concerned with cleaning the code up for storage in PROM, and pertain to FOCAL 65-E for the KIM-1, obtained from the 6502 Program Exchange in Reno, Nevada.

The first order of business in preparing FOCAL for PROM is to get rid of the self-modifying parts. The three places I found where FOCAL modifies itself in the main code are at locs. $2348-2353, $282C-283D, and $3408-3414. A fourth place occurs in page zero, where it doesn't matter since page zero is always RAM in 6502 systems. The other places are easily fixed. I borrowed a few locations from an obscure corner of KIM's on-board RAM to do it; neither KIM nor FOCAL seems to mind. The changes are as follows:

```
2348 was 8C 52 23 change to 8C DE 17 STY DJADR
234E was 8C 53 23 change to 8C DF 17 STY DJADR + 1
2351 was 4C 00 00 change to 4C DD 17 JMP $17DD
282C was 8C 3C 28 change to 8C DB 17 STY DJADR1
2835 was 8D 3D 28 change to 8D DC 17 STA DJADR1 + 1
283B was 6C 00 00 change to 4C DA 17 JMP $17DA
3408 was 8E 12 34 change to 8E E1 17 STX MOV11
340C was 8C 14 34 change to 8C E3 17 STY MOV22 + 1
3411 was B5 00 change to 4C E0 17 JMP MOVIT
3413 was 95 00 change to EA NOP
```

Additional code needed in page 17 is:

```
17DA 6C 00 00 JMP (0000)
17DD 4C 00 00 JMP 0000
17E0 B5 00 MOVIT LDA(X) 00
17E2 95 00 STA(X) 00
17E4 4C 15 34 JMP 3415
```

The address overwriting now occurs in page 17 RAM instead of in the main code, which can now be safely put into PROM.

Before doing so, however, we must relocate it. Note that relocation should not alter existing page boundaries (see warning on p. 44 of FOCAL 65-E Manual). This actually makes the job easier, because only the high-order bytes of addresses and address-words can be changed. Relocation then is accomplished by (a) adding the desired offset to the third byte of all three-byte instructions which do not reference page zero; (b) Offsetting the data words for routines such as PUSHJ and POPJ, the software stack manipulators. These words are scattered here and there through all the code. A listing of their high-order halves is given in Table 1; they are address words, so only the second byte is to be offset. (c) Offsetting the high-order bytes of the address tables at the end of the FOCAL code, which are at hex locs. 34FA-3515, 3546-3557, 356A-356E, 3598-359C, 35A2-35A6, 35AC-35B0, 35B6-35BA, 35C0-35C4, and 35CA-35CE. (d) Adding the offset to the IRQ-vector initialization byte at loc. 34AE (I date your cleverest relocation program to find *that* one!).

A final change necessary to execute FOCAL from PROM is to change the RAM allocation for program statements and variables so it is located in RAM, instead at the end of the machine code to go in PROM. The original start of this allocation is at loc. 35F3, but if you are going to PROM your FOCAL I suggest you save some PROM locations by deleting the heading that is printed as if it were line number 00.00 by the Write command. I retained only the line number zero and a carriage return in my version, since the program expects to print something there. This saves twenty-seven bytes of memory. In my system, I decided to start the RAM storage for statements and data at loc. 4000, so initialization there is as follows:

**4000 00 ;line number**
**4001 00 ;of 00.00**
**4002 0D ;ASCII 'CR'**
**4003 FE ;PBEG**
**4004 FF ;VEND**

To tell FOCAL where to put its statements and variables, some page zero locations need to be changed:

```
002F was D4 35 change to 00 40 ;beginning of RAM allocation
0031 was F2 35 change to 03 40 ;start of user's text
003E was F3 35 change to 04 40 ;start of variable list
0040 was F3 35 change to 04 40 ;start of variables for "erase all"
0042 was F3 35 change to 04 40 ;end of variable list
```

The code to accomplish page zero and page 17 setup and initialize the user RAM is given in Table 3. The code begins at loc. 3677 instead of right after the FOCAL code because I have some other modifications in between; the user will want to relocate this to suit his system anyhow.

[1]See 6502 User Notes, issue #16, and errata in issue #17.

**Table 1. Table of High-Order Data Bytes Used by POPJ and PUSHJ. Add Offset to Relocate.**

| Hex Location | Original Contents |
|---|---|
| 2088 | 23 |
| 20B2 | 23 |
| 20D7 | 29 |
| 212F | 21 |
| 219E | 21 |
| 21D0 | 23 |
| 21FE | 23 |
| 2440 | 2B |
| 2452 | 29 |
| 24BB | 29 |
| 2502 | 2B |
| 2516 | 29 |
| 2533 | 29 |
| 2546 | 29 |
| 256A | 29 |
| 257A | 23 |
| 25EB | 29 |
| 29DC | 29 |
| 29E5 | 2D |
| 2A45 | 2B |
| 2A5D | 29 |
| 2ABE | 29 |
| 2B97 | 29 |
| 2EFF | 29 |
| 2F7F | 29 |
| 2FA3 | 29 |
| 2FE8 | 29 |
| 300D | 2B |
| 309E | 29 |
| 316A | 2B |
| 3186 | 29 |
| 31A8 | 21 |
| 34AE | 2C |

**Table 2. FOCAL Initialization**

```
3677  A2 00      COLDST  LDX $00       ;Initialize table & instructions
      BD A0 36   LOOP1   LDA(X) TABL1  ;at page zero
      95 20              STA(X)
      E8                 INX
      E0 BD              CPX $BD       ;Initialize $17BA-$17E6 for
      D0 F6              BNE LOOP1     ;removal of self-modifying code
      A2 00              LDX $00       ;in FOCAL
      BD 5D 37   LOOP2   LDA(X) TABL2
      9D DA 17           STA(X) $17DA
      E8                 INX
      E0 0D              CPX $0D
      D0 F5              BNE LOOP2     ;Initialize User RAM
3690  A2 00              LDX $00       ;with line number
      BD 6A 37   LOOP3   LDA(X) TABL3  ;zero and data bytes
      9D 00 40           STA(X) $4000
      E8                 INX
      E0 05              CPX $05       ;Go to FOCAL cold start
      D0 F5              BNE LOOP3     ;page zero constants & code
369D  4C 00 20           JMP FOCAL
36A0  ( contents of         )
  :   { FOCAL locs. $0020-  } TABL1
  :   ( $00DC go here       )          ;Table for patches to
375C                                   ;remove self-modifying
375D  6C 00 00   TABL2                 ;code in FOCAL
      4C 00 00
      B5 00
      95 00
      4C 15 34                         ;Line no.
367A  00         TABL3                 ;of 00.00
      00                               ;ASCII 'CR'
      0D                               ;PBEG
      FE                               ;VEND
367E  FF
```

©

# COSMAC QUICKIES

Jess Hillman

Quick, inexpensive solutions to control problems are always desirable, so owners of COSMAC Elf microcomputers may find many interesting ways to use the "quickie" programs in listings one, two and three accompanying this article.

The programs were written specifically for my Quest Super Elf, which has 4.25K RAM, but they should run with very little tweaking on any 1802-based system, if entered beginning at any quarter-K page boundry.

Listing one is an interval timing program that can be set for any delay from a couple of seconds to about ten minutes by varying only the data byte in location 0011. By changing the program beginning at location 0015 to read: "9F FB XX (any value from 00 to FF) CE 30 05 7B 30 00" the program can set intervals up to two days (actually the maximum value for Register F falls a few minutes short of 48 hours). The data bytes in locations 0011 and 17 set the final value selected. Since the 1802 has plenty of registers for such usage, it would be very easy to establish intervals months long.

The program specifically uses Register E, one of the 1802's sixteen, sixteen-bit general purpose registers, as a timer that continually counts down from hex FFFF. When Register E reaches zero, a fact discovered by testing both high and low bytes, Register F is incremented by one. The F register is then tested to see if the predetermined value has been set. If not, the timing loop continues.

Once the proper value for F has been reached, the 1802 sets its Q line, an external flag that can be set or reset depending on various internal conditions of the processor, to a logic "1." After thus acknowledging it has reached the required time, the Q line is reset to logic zero and the timer resumes its labors. The Q line transition can be latched by connecting it to an integrated circuit such as the 74LS175 or the CMOS 4016, and held for use in driving a transistor, opto-isolator or relay (for high-voltage uses) to operate a coffee pot, television, stereo--practically anything controllable with an electronic switch.

Newcomers to the 1802 be warned: when tying to the Q line always buffer it generously with an IC like the 4050 or 4049, either of which can drive two TTL loads. Otherwise, you risk ruining your microprocessing chip.

A variation of this use of the Q line is found in listing two, in which the operator wishing access to the Q line must first enter three predetermined, two-hex-digit numbers into memory in the proper sequence. That oughta keep Pop's pet project safe from the kids!

As the data bytes for the "combination" are entered into memory, the 1802 performs a logical exclusive or with each byte in turn, using data stored at addresses 000D, 0017 and 0021 respectively. If the wrong number is entered at any point, the program jumps to the error subroutine beginning at location 0030, which momentarily outputs an "EE" to the data display (I have seven-segment LEDs) while executing a three-second timing delay, then outputs a "00" to the data display and jumps back to the beginning of the program.

In listing two, once the proper number sequence has been entered, the Q line goes high and stays that way until the input key is pressed and released (or external data flag EF4 is otherwise pulled low). Once EF4 goes low and returns to its normal state, Q goes to logic zero and the program loops back to the beginning again.

As written, you would have to enter 05 (at 0D), 17 (at 17) and 98 (at 21) to turn the Q line from logic low to logic high. You can change the data bytes for any combination you wish. The chances of someone solving the combination decrease if you add more numbers to the combination.

Listing three changes this program to utilize an output port and eight data bits to control various devices. When entered as listed and run, the program will: require you to enter the three number combination properly, after which the Q line goes high (on my system this turns on an LED); then you must enter a status byte which will be put in the memory stack and also latched into the output port (1802 output instructions are 6N, where N designates a port from one to seven). I use a 63 instruction because that port is readily available on my Elf's expansion board. Once the status byte has been latched to the output port, the program loops back to the beginning of memory and starts again.

The status byte can be whatever you want it to be, depending on your interface configuration. Eight data lines are immediately available, so using transistors, relays or a combination of techniques can give you immediate computer control of the major energy consuming devices in your home--air conditioning, hot water heater, and so on.

By expanding the interval timer to include a lookup table of status bytes for dispatch to the output port at various times of the day, automatic control your home's major functions becomes possible. The only question you must answer is how elaborate you want it to be.

Using the upper four bits of the status tied to, say, a 74LS154 four-to-sixteen line decoder, with the lower four bits or-tied to sixteen latches like the 74LS175, it would be possible to control up to sixty-four devices from your micro's output port.

Possible expansions and combinations of these programs are virtually endless. As quickie programs go, however, they should give newcomers to the

1802, or people struggling with a system they've had for a time, a feel for register manipulation and control application possibilities of the typical COSMAC system.

| Listing 1-- Interval Timer | Address | Data | Mnemonics | Comments |
|---|---|---|---|---|
| | 0000 | F8 00 | LDI 00 | Initialize Reg. F for |
| | 02 | AF BF | PLO, PHI | use as workspace |
| | 04 | 7A | REQ | Make sure Q is at logic "0" |
| | 05 | 2E | DEC R.2 | Decrement timer |
| | 06 | 9E CE | GHI, LSZ | Check timer, long skip if zero |
| | 08 | 30 05 | BR 05 | If not zero, continue loop |
| | 0A | 8E CE | GLO, LSZ | If high byte zero, check low byte |
| | 0C | 30 05 | BR 05 | If not zero, continue loop |
| | 0E | 1F | INC Reg. F | If low byte zero, increment workspace register by one |
| | 0F | 8F | GLO R.F | Get new value from Register |
| | 10 | FB 17 | XRI 17 | Exclusive Or with predetermined value |
| | 12 | CE | LSZ | If values match, long skip (PC incremented by 2) |
| | 13 | 30 05 | BR 05 | If no match, continue loop |
| | 15 | 7B | SEQ | Set Q line at logic "1" |
| | 16 | 30 00 | BR 00 | Then start looping again |
| | 17 | 00 | IDL | End |

| Listing 2-- Combination Lock | Address | Data | Mnemonics | Comments |
|---|---|---|---|---|
| | 0000 | F8 00 | LDI 00 | Set up workspace in memory |
| | 02 | B4 | PHI | using Reg. 4 to point to |
| | 03 | F8 F0 | LDI F0 | stack beginning at |
| | 05 | A4 | PLO | address 00F0 |
| | 06 | E4 | SEX | |
| | 07 | 3F 07 | BN4 07 | Loop if EF4 equals zero |
| | 09 | 37 09 | B4 09 | Loop if EF4 equals one |
| | 0B | 6C | INP 4 | Get keyboard byte |
| | 0C | FB 05 | XRI 05 | Check if correct combination # |
| | 0E | CE | LSZ | Long skip if zero (a match!) |
| | 0F | 30 30 | BR 30 | Else go to error subroutine |
| | 11 | 3F 11 | BN4 | Wait until next byte latched |
| | 13 | 37 13 | B4 | in from keyboard |
| | 15 | 6C | INP 4 | get the byte |
| | 16 | FB 17 | XRI 17 | If it matches, too, then long skip |
| | 18 | CE | LSZ | |
| | 19 | 30 30 | BR 30 | Go To error subroutine otherwise |
| | 1B | 3F 1B | BN4 | If second number matches, wait |
| | 1D | 37 1D | B4 | for last combination number |
| | 1F | 6C | INP 4 | Get it |
| | 20 | FB 98 | XRI 98 | See if it, too, matches |
| | 22 | CE | LSZ | Long skip if it does |
| | 23 | 30 30 | BR 30 | Error subroutine if it doesn't |
| | 25 | 7B | SEQ | All numbers OK, Q equals "1" |
| | 26 | 3F 26 | BN4 | Keep Q line on until input |
| | 28 | 37 28 | B4 | key pressed and released |
| | 2A | 7A | REQ | Then turn it off and go back |
| | 2B | 30 00 | BR 00 | to beginning |
| | | | | **Error subroutine** |
| | 30 | F8 EE | LDI EE | Load message "EE" |
| | 32 | 54 | STR | store it in stack, then |
| | 33 | 64 | OUT 4 | output to LED display port |
| | 34 | BF | PHI Reg. F | Also store in Register F |
| | 35 | 2F | DEC Reg. F | Reg. F decremented by one |
| | 36 | 9F | GHI | Check byte in Reg. F |
| | 37 | CE | LSZ | Skip next two bytes if zero |
| | 38 | 30 35 | BR 35 | Otherwise loop |
| | 3A | F8 00 | LDI 00 | Load "00" and output to |
| | 3C | 54 | STR | clear error message from |
| | 3D | 64 | OUT 4 | display |
| | 3E | 30 00 | BR 00 | Go back and try again |

| Listing 3-- Controlling multiple devices | Address | Data | Mnemonics | Comments |
|---|---|---|---|---|
| | 002A | 6C | INP 4 | Get the byte input after correct combination given |
| | 2B | 54 | STR | Put status byte in stack |
| | 2C | 63 | OUT 3 | Output status byte to device interface |
| | 2D | 7A | REQ | Turn Q off |
| | 2E | 30 00 | BR 00 | Then go back to start |

©

# The 1802 Instruction Set

Dann McCreary
Box 16435
San Diego, CA 92116

In case you missed our first column, we took a flight of fancy over the 1802 to survey it's architecture. With this installment we begin a leisurely look at the 1802s' instruction set. Where possible, we'll try to compare and contrast 1802 instructions with similar instructions on the 6502.

Before we get rolling, here's an interesting bit of news I got from a certain OEM user of the RCA 1802. They have been delivering 1802 based systems which run at a clock rate of 3.2 MHZ. The systems were designed well within RCA's specs for the 1802. They are real-time systems and much of the software developed for them depends on that clock rate. Well, it seems that RCA has since had second thoughts about their 1802 speed spec. They've notified their customers that the 1802 is now only useable at up to 2.5 MHZ. Not nice! If any other 1802 users out there feel like victims of the "sting", I'd like to hear about it.

Do you remember the 1802s' I and N registers? A large number of 1802 instructions can be readily understood by breaking them down into their I and N components. Generally speaking, the contents of I determine the operation to take place, while the contents of N designate the general purpose 16 bit register to be used or affected. Look at the illustration of the instruction matrix. Each unbroken horizontal area represents an instruction of this type. Even the LDN instruction in the first row is like this except that it is not applicable to R0, since the 00 hex op-code has been preempted for use as the IDL instruction.

Let's take for example a 1C hex. The 1 in the I register says that this is an increment instruction and the C in the N register says that it is to affect RC, one of the 16 bit registers. You could change this to affect any register merely by changing the N portion of the op-code. (By the way, this kind of consistency across the board makes the 1802 one of the easiest processors to hand assemble code for.) INC performs a true 16 bit increment with rollover such that FFFF hex increments to 0000 hex. It is very much like the 6502s' INX and INY instructions except that X and Y are only 8 bit registers. One caution: since the 1802 has no status register, the only way to branch on the result of an INC is to test the register contents by moving them in to the 1802 accumulator, register D. Also note that the 6502 INC lets you increment memory contents but the 1802 INC is strictly for registers. These comments apply as well to the DEC instruction, 2N hex (where N represents any hex digit), but in the opposite direction.

On the 6502 you can load the accumulator (LDA) from memory using a variety of addressing modes. The 1802 gives you LDN (0N hex) which lets you load D with the contents of a memory location, and LDA (Load Advance, 4N hex) which is a LDN and an INC rolled into one. But which memory location does the data come from? The contents of the register designated by RN go out on the address bus, selecting a memory location. Right there, my friend, is what can be one of the more frustrating aspects of programming an 1802 - the only way to access memory is via a register. This implies doing some LDIs (Load Immediates, identical in function to the 6502 Immediate Mode) to set up a memory address. Compare these:

```
                 6502
AD 3412 LDA ADDR1 .. READ MEMORY
                 1802
F8 12 LDI A.1 (ADDR1) .. SET UP
B8 PHI R8 .. R8 TO POINT TO
F8 34 LDI A.0 (ADDR1) .. THE DESIRED
AQ8 PLO R8 .. MEMORY ADDRESS
08 LDN R8 .. READ MEMORY
```

Looks pretty bad for the 1802, right?
Well, consider that every time you want to load or store at that memory location with a 6502 it's going to take 3 BYTES. On the 1802, once you've set up the address you can load or store (STR) with only one BYTE. The 1802 LDA instruction also gives you 16 bit auto-indexing memory access with a one BYTE instruction! To do this on the 6502 you would have to set up two page zero memory locations with the starting address and then do a double-precision increment between each memory access. I'll let you figure out how many BYTES that would take you!

In our example we also used PHI and PLO. These instructions transfer the contents of the accumulator, D, to the high or low order BYTE of the register designated by N (RN). The converses of PHI and PLO are GHI and GLO. GHI transfers the most significant BYTE of RN to D. GLO transfers the least significant BYTE of RN to D.

The last two instructions that operate consistently across the board are SEP and SEX. SEP simply takes the contents of N and places them in the P register. You may recall that the 4 bit P register determines which 16 bit register is the current program counter. At system reset, P is forced to zero, making R0 the initial program counter. After initializing some registers, a program might execute a SEP instruction as a simple means of transferring to a subroutine. Let's consider a subroutine with the simple task of toggling the Q flip-flop from its present state to the opposite state. We'll assume that our main program is using R0 as program counter and that the subroutine will be at 1101 hex. Before we can use the subroutine, we must put its' starting address into a register like this:

```
F8 11 LDI A.1(QSUB) .. MSB OF ADDRESS
B7 PHI R7 .. INTO R7.1
F8 01 LDI A.0(QSUB) .. LSB of ADDRESS
A7 PLO R7 .. INTO R7.0
```

The .1 and .0 notations indicate HI and LO order portions of a register or 16 bit address. Now we can execute this:

```
D7 SEP R7 .. GO TO QSUB
```

When the SEP R7 is executed, R0 is left pointing at the instruction immediately following the SEP R7. So can you guess what dur return from subroutine instruction will be? Right, a SEP R0! Here's QSUB:

```
1100
1101
D0 QRET: SEP R0 .. RETURN TO MAIN PROGRAM
CD QSUB: LSQ .. IF Q IS SET, SKIP 2 BYTES
7B SEQ .. SET Q
38 NBR .. SKIP A BYTE
7A REQ .. RESET Q
30 00 BR QRET .. GO RETURN
```

Why is the return at the top? After executing QSUB and branching to QRET, the last thing the subroutine does is execute the SEP R0. As that is executed, R7 is incremented and now points once again to QSUB, ready for the next subroutine call. Please note that only programs using R0 as their program counter may call this subroutine. Incidently, we used some new instructions here - SEQ and REQ which are considered control type instructions and some branch and skip instructions. The branch instructions are the unconditional branch, BR, and the unconditional branch not, NBR. Unlike the 6502 which provides relative branching, the 1802 only permits absolute branching. Its short branches are much like a 6502 JMP instruction, only the high-order address is whatever is currently in the program counter. Thus you can only do a short branch within the memory page you are already on. Also, when you relocate code in memory, most of the branch addresses must be changed - quite tedious if you are assembling the code by hand. The 1802 *does* allow unconditional short branches, something that sure would be nice to have on the 6502. The NBR instruction is interesting because it effectively results in a skip and in fact you may wish to use the mnemonic skip rather than NBR. Conditional branches are possible based on the state of Q or DF and also on the zero/non-zero state of D. This includes both short and long branches. The long branches are like the jump on the 6502 but if you are assembling by hand, beware! All 16 bit addresses on the 1802 are specified in normal order, not reversed as on the 6502. Conditional short branches are also possible based on the state of the external flag lines, EFI-EF4. As an exercise, try writing a subroutine that returns a number from 1 to 4 in D, based on which one of four external flag lines is activated. Use the instructions B1, B2, B3, and B4 to branch on the state of the flags.

# 1802 INSTRUCTION MATRIX

| I \ N | 0 | 1 – 3 | 4 | 5 – 7 | 8 | 9 – B | C – F |
|---|---|---|---|---|---|---|---|
| 0 | IDL | LOAD VIA RN (LDN) | | | | | |
| 1 | INCREMENT RN (INC) | | | | | | |
| 2 | DECREMENT RN (DEC) | | | | | | |
| 3 | BRANCH ON Q, Z, DF | | BRANCH ON EF1-EF4 | | BRANCH NOT ON Q, Z, DF | | BRANCH NOT ON EF2-EF4 |
| 4 | LOAD-ADVANCE RN (LDA) | | | | | | |
| 5 | STORE VIA RN (STR) | | | | | | |
| 6 | IRX | OUTPUT | | | X | INPUT | |
| 7 | CONTROL & MEMORY REF. | | ARITHMETIC W/CARRY | | CONTROL | | ARITH. IMMEDIATE W/CARRY |
| 8 | GET LOW BYTE OF RN (GLO) | | | | | | |
| 9 | GET HIGH BYTE OF RN (GHI) | | | | | | |
| A | SET LOW BYTE OF RN (PLO) | | | | | | |
| B | SET HIGH BYTE OF RN (PHI) | | | | | | |
| C | LONG BRANCH | | NOP | LONG SKIP | LONG BRANCH | | LONG SKIP |
| D | SET P REGISTER (SEP) | | | | | | |
| E | SET X REGISTER (SEX) | | | | | | |
| F | LOGIC | | ARITHMETIC | | LOGIC IMMEDIATE | | ARITHMETIC IMMEDIATE |

The LSQ in our example is typical of the Long Skip isntructions. If the condition of the skip is met, the next two bytes are skipped. Otherwise, execution continues with the next instruction. Long Skip conditions are the same as for Long Branches, with the addition of LSIE which permits testing the state of the interrupt enable flag, IE.

Finally, we come to the instruction we've all been waiting for- SEX! Obviously the designers at RCA are not your staid, single-minded, no-nonsense engineering types at all. They appear to enjoy mixing a little fun in with their work! Setting X, from which the infamous mnemonic is derived, simply puts the value of N into the X register. A number of the instructions we have yet to look at interact with the memory location pointed to by the 16 bit register designated by X. We'll refer to that register as RX.

LDX is like LDN. It loads the contents of memory pointed to by RX into D. LDXA is to LDA as LDX is to LDN. STXD is like a STR, but instead of storing D via RN, it stores D at the location pointed to by RX and decrements RX to boot. Now if they had only included a load via X and decrement, and a store via X and advance (LDXD & STXA) the 1802 would have been much more versatile! Oh, well. So much for memory reference via RX.

The 1802 gives you a handful of arithmetic operations that let you subtract or add the contents of memory and D. Unlike the 6502, the 1802 gives you the option of excluding the carry (DF) from the operation. Thus, you can add without first clearing the carry and subtract without first setting the carry. Arithmetic can also be done using the immediate byte, allowing you to add or subtract fixed values.

The logical operators used VIA RX are AND, OR and XOR (Exclusive OR). They work much like their 6502 counterparts. They may also be used in immediate mode. D may be operated on also with the SHR, SHRC, SHL and SHLC. These are identical to the 6502 LSR, ROR, ASL and ROL commands, in that order except that they apply only to shifting the accumulator.

Input and output instructions are also intimately related to RX. To output a byte of data, you must first store it at the memory location pointed to by RX. RX is also incremented when the OUT is executed, making this handy for outputting messages from buffers. Using INP inputs a byte into D, but be careful! It also gets stored VIA RX. Be sure RX is pointing where you want it. When doing an Input or Output, the N lines on the 1802 chip are set to match bits 0, 1 and 2 of the contents of the N register. This makes I/O decoding in hardware somewhat simpler than the 6502 memory-mapped only approach.

IRX increments the X register. It is really a vestigial out instruction but no output is defined when N = 0. Note also that what might have been an INP 0 (68 HEX) is the one undefined 1802 opcode. It is now used on the recently released 1804 to add some features and correct some 1802 deficiencies.

Well, we've looked over all but a handful of the 1802's 255 instructions. All that are left are some control instructions with some relatively obscure or involved applications. We'll discuss them in later columns as we apply them. Meanwhile, try converting some 6502 code into 1802 code for practice. And let me hear some comments! I can be reached at Box 16435, San Diego, CA 92116. ©

# CAPUTE!

## Our Bug-Box

Robert Lock

Here, at long last, are the corrections to **Read Pet Tapes With Your Aim** that appeared in the March/April issue (#3) of COMPUTE.

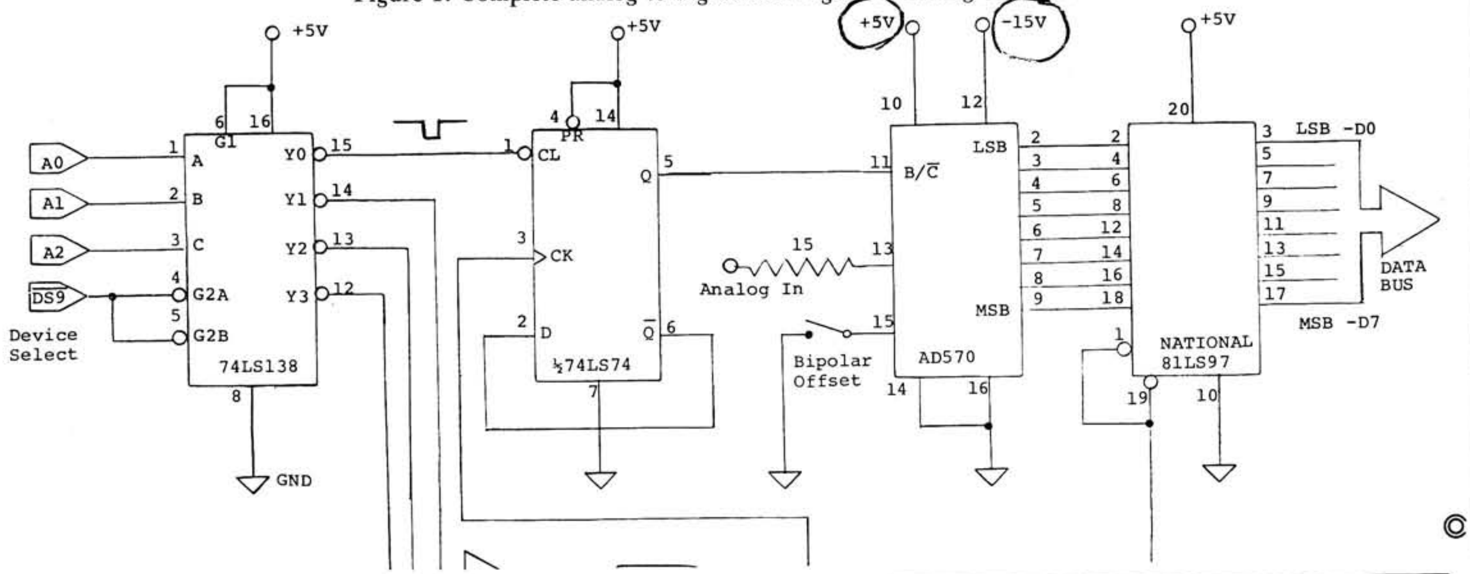
```
LINE # LOC      CODE          LINE

0041   0212   D0 F6                   BNE NEXT

0043   0214   20 89 03                JSR OFFON      ;TURN OFF TAPE
0044   0217   A2 00                   LDX #00        ;OUTPUT NAME OF FILE
0045   0219   BD 6F 04     NAME       LDA FILE,X
0046   021C   C9 20                   CMP #'         ;LOOK FOR BLANK AT END
0047   021E   F0 06                   BEQ LEN
0048   0220   20 7A E9                JSR OUTPUT
0049   0223   E8                      INX
0050   0224   D0 F3                   BNE NAME       ;GET NEXT LETTER
0051   0226   20 3E E8     LEN        JSR BLANK

0053   0229   18                      CLC            ;OUTPUT NECESSARY MEMORY
0054   022A   AD 6D 04                LDA END        ;FOR PROGRAM
0055   022D   69 62                   ADC #$62       ;ADD TO END
0056   022F   8D 6D 04                STA END        ;THE DIFFERENCE BETWEEN PET AND
0057   0232   AD 6E 04                LDA END+1      ;AIM BASIC START LOCATIONS
0058   0235   69 00                   ADC #00
0059   0237   20 46 EA                JSR NUMA       ;OUTPUT IT
0060   023A   AD 6D 04                LDA END
0061   023D   20 46 EA                JSR NUMA
0062   0240   20 3E E8                JSR BLANK

0064   0243   20 73 E9                JSR REDOUT     ;GET A CHARACTER
0065   0246   20 89 03                JSR OFFON      ;TURN ON TAPE
0066   0249   C9 59                   CMP #'Y        ;Y MEANS READ THIS FILE
0067   024B   F0 0B                   BEQ GO
0068   024D   8D 6A 04                STA FLAG       ;CHANGE FLAG'S VALUE
0069   0250   20 13 EA                JSR CRLOW
0070   0253   4C 0A 02                JMP NEXT       ;READ NEXT FILE ON TAPE

0072   0256   EA                      NOP            ;REMOVE IF USING
0073   0257   EA                      NOP            ;AN ASSEMBLER
ERRORS = 0000 <0000>
END OF ASSEMBLY
```

Note to you PET Owners who read COMPUTE. **Don't use** the Disk ID Changer program in Issue 5 until you see the **important** update in Issue 6. RCL

## Oops!

And here's an important correction for Marvin L. DeJong's compute II, Issue 2 schematic. (Page 6, Some **A/D And D/A Conversion Techniques.** Note that pins 10 and 12 (circled) are now (correctly) reversed.

Figure 1. Complete analog-to-digital and digital-to-analog converter circuit.



# Finally...Serious Expansion for the AIM-65


Introducing **Memory-Mate***, the AIM-65 expansion board that lets you spend your time on application solutions, not hardware hassles. Add Memory-Mate to your AIM-65 and make quick work of development and process control projects.

In its primary function, the Memory-Mate board provides 16-48K of RAM expansion assignable in 4K blocks anywhere in the system. Memory-Mate's parity check circuitry insures system RAM integrity (including AIM's 4K on-board RAM) for high reliability applications. The programmable write protect feature eases software development chores. This compact board, which fits directly **beneath** the AIM, also includes four programmable I/O ports, a tone generator for audible warnings, and sockets for 4K of PROM.

I/O intensive applications are accommodated with Memory-Mate's STD BUS interface option. Use off-the-shelf STD BUS cards to solve your biggest I/O problems.

The Memory-Mate with 16K RAM is priced at $475, with 16K expansion chip sets (including parity chip) costing $100 each. With 48-hour active burn-in and warranty for a full year, you won't have to worry about reliability either.

First of the complete AIM-Mate* series, Memory-Mate will be joined shortly by the Video-Mate*, Floppy-Mate* and the AIM-Mate case. For further information on the entire AIM-Mate series, write 'Attn: AIM-Mate Series' at the address below.

*TM Forethought Products

**Introductory Special for Compute II Readers**
**16K Memory-Mate Board for $399 thru Sept. 15! Call for details.**

Forethought Products

87070 Dukhobar Rd., Eugene, OR 97402
(503) 485-8575