Thoughts on the SWTP Computer System

Don't be a scrooge; catch the true spirit of this series with the HUMBUG monitor.

Peter A. Stark
PO Box 209
Mt. Kisco, NY 10549

This month, we continue the ROM monitor discussion we started last month. The first ROM monitor in SWTP systems was MIKBUG. Most software was designed to work with it, and so succeeding monitors have had to copy many of MIKBUG's routines and addresses.

The important MIKBUG entry points, which should be preserved in "compatible" monitors are:

8ADDR	E047—Input four hex digits into index register
BYTE	E055—Input two hex digits into A accumulator
OUTHL	E067—Output left BCD digit in A accumulator
OUTHR	E08B—Output right BCD digit in A accumulator
OUTCH	E075—Points to OUTEEE
INCH	E078—Points to INEEE
PDATA1	E07E—Print a text string pointed to by index reg.
INHEX	E0AA—Input a hex digit into A accumulator
OUT2H	E0BF—Output two hex digits pointed to by index reg.
OUT4HS	E0C8—Output four hex digits pointed to by index reg., followed by a space
OUT2HS	E0CA—Output two hex digits pointed to by index reg., followed by a space
OUTS	E0CC—Print a space
START	E0D0—Start MIKBUG
CONTRL	E0E3—Restart MIKBUG
INEEE	E1AC—Input a 7-bit character from keyboard
OUTEEE	E1D1—Output a character to terminal

These sixteen entry points are the major ones. In addition, there are about 20 more minor ones that you can include if you just copy most of MIKBUG, but which are probably otherwise not needed.

The one exception is the SWTP BILOAD program, which is used to speed up loading of binary tapes such as BASIC. This program uses these additional MIKBUG entry points:

DMPREG	E115—Print out CPU registers
LOAD19	E040—Part of load routine
SAV	E1A5
DE	E1F3
DEL	E1EF
IOUT2	E1E3

This loader does not work with an MP-S interface, so I chose not to include these entry points. However, I did include an entry point called INCH8 at E1F6, which is similar to INEEE except that it enters an 8-bit ASCII character rather than stripping off the parity bit to make it into seven bits, as INEEE does.

MIKBUG also uses the 128-byte scratchpad RAM starting at location A000. There are some differences, however, between MIKBUG and SWTBUG in address assignments in this area, and I chose to go with SWTBUG here rather than with MIKBUG. The important addresses are as follows:

IRQ	A000—IRQ interrupt vector
BEGA	A002—Beginning address for punching, etc.
ENDA	A004—Ending address for punching, etc.
NMI	A006—NMI interrupt vector
SP	A008—User stack pointer
PORADD	A00A—Address of the control port in use
PORECH	A00C—Terminal echo on/off flag
XHI	A00D—High-order half of index register
XLOW	A00E—Low-order half of index register
CKSM	A00F—Checksum
SWIJMP	A012—SWI interrupt vector
PC	A048—Program counter for Go command

MIKBUG had XHI and XLOW one location lower, and some other monitors (as well as some user software) go along with this convention.

I also treated the stack differently. MIKBUG and SWTBUG always initialize the stack when they are started up at A042 and down. The G command then loads the next seven bytes into CPU registers and jumps to a user program with the stack pointer pointing to A049. So, in a way, we can think of the area below A042 as being a monitor stack, while the area just below A049 is a user stack.

But SWTBUG's J command doesn't change the stack pointer when going to a user program; it leaves it pointing to the monitor area. Likewise, when a breakpoint is encountered, it leaves the stack pointer unchanged when it executes its own routines. This results in some weird occurrences when the monitor and user stacks wipe each other out. It becomes even more interesting when you consider that some user software initializes the stack elsewhere . . . such as at A042.

Because of this, I put other HUMBUG storage locations in a separate RAM—far away from the MIKBUG/SWTBUG RAM—and treated the stacks differently. The monitor stack is now always at D07F. A jump command always goes to a user program with the stack pointer at A07D (with a return address at A07E/F, so jumping to a subroutine will result in a return back to the monitor), and a GO command always goes to the user program with the stack pointer at A049.

This keeps monitor and user stacks completely separate so they never clobber each other. It does require a separate RAM, however, at locations D000-D07F for strictly monitor use. In return, it keeps HUMBUG storage strictly compatible with any stack or storage assignment made by other programs, so there is never a problem.

In my system, the storage at D000 is provided by the 4K board I mentioned earlier. In two other systems that are currently running under HUMBUG, the memory is provided by the CPU board's 6810, relocated to C000-DFFF as also mentioned last month.

I/O Control from the Keyboard

HUMBUG's control terminal is a serial terminal using an MP-S card at port 1, which provides all input to the monitor,

and also standard output. Location PORECH (A00A) contains $8004, which points to this port. By changing this number, you can redirect the control port to an MP-S card at any other port. (I'm describing the common version of HUMBUG; my own has its I/O at $F804.)

In addition, HUMBUG can provide an output to a second MP-S at port 0, to a user-written output routine in another EPROM or (in the 3K 2708 version) to the Percom video board.

Any time that the monitor is looking for commands or any time that INEEE or OUTEEE is called, HUMBUG checks this port for a control-S break character arriving from the keyboard. When a control-S is detected, HUMBUG echoes with a bell (control-G) and halts all current I/O.

When I/O is halted, HUMBUG waits for one more character, which is used for controlling monitor ports. If it is received by INEEE, then it is not returned back to whatever program called INEEE. This provides control of output ports without upsetting other programs. This control character can be one of the following:

CR—cancels the current program and usually does a return to the monitor. But the return is handled through a pointer in RAM, so that other programs could change the pointer and force a return to themselves.

0—turns port 0 on and off.

1—does the same for port 1.

D—does the same for a user-written port routine.

P—turns the pause feature on and off. When the pause feature is on, output will stop every 16 lines to allow it to be read when using CRT terminals.

Any other character is ignored. The 0,1 and D characters toggle their corresponding ports; if a port is on then it goes off, if it is off then it goes on. Since these characters are not echoed or even returned to calling programs, ports can be turned on and off in the middle of input or output.

The video board output normally runs all the time and is not controlled. (There is a flag in monitor RAM, however, that disables it if I want to use it for graphics or memory-mapped output.)

When another port is on, then the video output simply runs at the speed of the slowest port. But when all other ports are turned off, then the video board runs at breakneck speed, limited only by CPU speed.

This feature is extremely versatile. Not only does it allow precise printer control, but it also permits rapid skipping ahead at video speed. (The 2K 2716 version, which does not support the video board, will skip ahead even faster when you turn off all output.) Moreover, the control-S/CR combination allows you to abort jammed programs without reaching for the RESET button.

Extended Debugging Facilities

My third requirement for improved debugging power was met in several ways. First, the HD command allows a hex dump of selected memory areas. The DE command prints a "desembled" listing of machine code, formatted by address and instruction. Thus, a DE dump of a program might go like this:

```
1000 86 41
1002 BD E1D1
1005 4C
1006 B7 1103
```

An AO command outputs memory data in ASCII so I can scan for strings. An FM command allows filling memory with a specified byte. This is convenient to fill memory with 3F (SWI) instructions to catch programs that go wild. The FI command allows searching memory to find one, two or three bytes. The MO command moves memory contents from one place to another, even if the new area overlaps the old area.

But the most important function is the breakpoint and single-step facility. Up to four breakpoints can be set in programs, and whenever a program encounters such a breakpoint (or any SWI instruction anywhere), an interrupt returns control to HUMBUG, which then prints out the register contents

and stops. HUMBUG keeps track of breakpoint locations and the instructions existing in those locations and prints a listing of them whenever the BP (breakpoint print) command is given. This reminds you where you have put the breaks. An important feature is that HUMBUG doesn't forget about them either when a jump back to the monitor is done, or when RESET is pressed.

SS is used for single-stepping through programs. Each time you type SS, HUMBUG prints out the address and code of the next instruction, executes it and then prints out the contents of all registers after the instruction is completed. It will single-step all instructions except WAI, SWI and RTI, and cannot single-step into or through ROM. HUMBUG prints out NO! whenever any of these are attempted.

FCROM

FCROM occupies addresses FC00-FFFF. It contains the reset and interrupt vectors that the 6800 CPU needs at locations FFF8-FFFF. So, without this ROM, the system cannot function at all.

FCROM contains all of the common MIKBUG I/O routines. But since this ROM is at the end of memory, none of these routines are at MIKBUG-compatible addresses. Instead, they are simply consecutively placed wherever they fit. To allow future changes, though, they are vectored through a jump table that starts at FC00:

```
FC00 JMP COLDST
FC03 JMP WARMST
FC06 JMP HOTST
FC09 JMP INEEE
FC0C JMP OUTEEE
etc.
```

Even when FCROM is changed in the future, these pointers will stay in the same place, and so external jumps into FCROM will stay unchanged.

OUTEEE and INEEE provide all of the port control features mentioned before. In addition, FCROM has a command processor that accepts monitor commands from the keyboard and processes them. But it only recognizes two commands—ME for memory examine and change and JU to jump to a user program. These are the absolute minimum that the monitor could have and still work.

Monitor Extendability

My fourth major requirement was to allow the monitor to be changed or expanded without too much work. As it now stands, I can add EPROMs without changing the existing ones. Moreover, I can even unplug some of the existing EPROMs from the system, and the rest of the monitor will still work! (Since the 2716 version consists of just one EPROM, this obviously doesn't apply to it.)

The 2708 version of HUMBUG consists of three 1K EPROMs: FCROM, E0ROM and E4ROM.

FCROM is completely self-contained and will run all by itself, even when the other EPROMs are unplugged. It contains all port control and video board control and, with the ME and JU commands, can load and execute other programs. But it is obviously limited; it relies on the other EPROMs in the system. It also doesn't have MIKBUG-compatible entry points, although it does have all the required routines.

This is where the extendability feature comes in. Notice in the above table that there is an entry point at FC00:

```
FC00 JMP COLDST
```

This is the main entry point when you first turn the system on or when you push RESET. This is a "cold-start," which initializes ports 0 and 1 and initializes the video board.

Once this is done, the FCROM program checks to see whether there is a ROM starting at address E000. If there isn't, then it proceeds with a "warm-start" initialization, where the program turns on port 1, turns off other ports and sets more registers. But if it detects that there is a ROM at E000, it executes a JSR to that ROM before doing the warm-start. This gives E0ROM a chance to execute a cold-start too.

When E0ROM is finished with

```
                * INTERRUPT VECTORS
FFE8 FE A000  IRQV   LDX   IRQ       IRQ VECTOR VIA A000
FFEB 6E 00           JMP   0,X
FFED FE A012  SWIV   LDX   SWIJMP    SWI VECTOR VIA A012
FFF0 6E 00           JMP   0,X
FFF2 FE A006  NMIV   LDX   NMI       NMI VECTOR VIA A006
FFF5 6E 00           JMP   0,X
   (FFF8)            ORG   $FFF8
FFF8 FF E8           FDB   IRQV IRQ VECTOR
FFFA FF ED           FDB   SWIV SOFTWARE INTERRUPT
FFFC FF F2           FDB   NMIV NMI VECTOR
FFFE FC 00           FDB   COLDV RESTART VECTOR FOR RESET
```

Listing 1. Interrupt and Reset vectors.

its cold-start, it checks for the presence of a ROM at either E400 or E800; if it detects one, it jumps there. Each EPROM gets its chance at a cold-start initialization. If E4ROM is installed at E400, it gets control; if not, then control either goes to the next ROM (if any) or returns to FCROM. Initialization is divided into cold-start and warm-start, and each of these transfers control from ROM to ROM.

When all initialization is completed, FCROM takes over again and looks for a command. If an ME or JU command is entered, then FCROM executes a memory change or jump itself. Otherwise, it puts the two command characters into accumulators A and B and transfers control to other ROMs, in turn. If one of these recognizes a valid command, it executes it; otherwise, control goes to the next ROM. Ultimately, control passes back to FCROM.

Passing control back and forth between ROMs allows more ROMs to be added at any time. Moreover, if one ROM is unplugged, the remaining ROMs still get control and can still execute their own commands. In this way, you can expand or modify HUMBUG without reburning all three EPROMs. But there is a price to be paid: an additional amount of housekeeping in each EPROM, which takes up about 40 bytes.

E0ROM

E0ROM, the second 2708, is at locations E000-E3FF. Although the system will run with just FCROM, E0ROM is essential for MIKBUG compatibility because the E0ROM has sixteen MIKBUG-compatible jump vectors that point to the corresponding locations of FCROM. For instance, location E1AC of E0ROM contains an instruction that says JMP to $FC09, which is the actual entry point for INEEE in FCROM. Each MIKBUG entry point has such a JMP.

This is a different approach from SWTBUG and other monitors, which simply put these routines at the same addresses as MIKBUG did and then try to fit everything in. Here all the routines are elsewhere, and only JMP instructions exist.

Woven in between these JMPs are the cold- and warm-start routines, the command processor that recognizes monitor commands and routines for the following commands:

LO—Load MIKBUG-formatted tape
PU—Punch/Save MIKBUG-formatted tape
EN—Punch end-of-tape with program counter and S9 code
FD—Bootstrap for Flex disk
PD—Bootstrap for Percom disk (or go to C000 EPROM)
GO—Go to user program using A048/A049 address
CL—Clear terminal screen
FI—Find one, two or three bytes in memory
HD—Formatted hex dump of memory
FM—Fill memory with a byte
CS—Compute a 16-bit checksum of memory contents
MT—Perform a memory test
PC—Print contents of A048/A049 (program counter).

Thus, this ROM puts in all of the necessary routines to make HUMBUG compatible with user programs and also puts in all the common SWTBUG commands, except breakpoints and register examine.

E0ROM has one more routine—FROMTO. This routine essentially asks for an input from the keyboard of a FROM address and a TO address, which are placed into BEGA location A002 and ENDA location A004 in the monitor scratchpad. This routine is called by most other commands to specify the beginning and end of desired memory.

The PU command is a good example. In SWTBUG or MIKBUG, locations A002 and A004 had to be preset to the starting and ending locations before calling the P command. In HUMBUG, the PU command uses the FROMTO routine to ask for the beginning and ending locations. This routine is set up so that entry of a carriage return will make it use the previous values.

E4ROM

E4ROM is the third 2708 and occupies addresses E400 through E7FF. Its cold-start and warm-start initialization and passing control to the next ROM are similar to those of E0ROM, but E4ROM adds the following commands:

DE—Desemble memory and print machine-language codes
BP—Breakpoint printout
BR—Breakpoint set or reset
CO—Continue after a breakpoint
RE—Register examine
SS—Single-step
AI—ASCII input into memory
AO—ASCII output from memory
MO—Move memory contents

The exact functions of these will become clear when we examine the actual programs.

Since the system is set up to allow more ROMs to be easily added, there are obviously others available. E8ROM, for instance, adds commands to compare memory contents, change terminal baud rate from the keyboard and change control ports. But these are just frosting on the cake, not really needed for most systems.

Let's examine some of the actual HUMBUG code.

Initialization and Reset

A 6800 requires four address vectors to be located in the top eight memory locations, FFF8 through FFFF, which are used to vector resets and interrupts. These four vectors are:

FFF8 and FFF9—IRQ vector
FFFA and FFFB—SWI vector
FFFC and FFFD—NMI vector
FFFE and FFFF—Reset vector

When you press the reset button or when an interrupt occurs, the 6800 pulls the appropriate address out of one of these four locations and puts it in the program counter. This causes a jump to that address. For that reason, when the system is first turned on, at least the reset vector and the routine it points to must already be in memory. This is why every 6800 system has its ROM located at the very top of memory.

Listing 1 shows the portion of HUMBUG's FCROM that contains the very top of memory. FFF8 through FFFF contain these four vectors: IRQ points to FFE8, SWI points to FFED, NMI points to FFF2 and reset points to FC00. Thus, when a reset is completed, the 6800 starts executing from location FC00, which is the beginning of FCROM.

The interrupt vectors all point to locations in ROM, shown just above that. When an interrupt occurs, the computer goes to the appropriate routine, loads a number from RAM into the index register and then does an in-

```
                * JUMP VECTORS
   (FC00)               ORG   $FC00
FC00 7E FC33  COLDV     JMP   COLDST    COLD START ENTRY POINT
FC03 7E FC52  WARMV     JMP   WARMST    WARM START ENTRY POINT
FC06 7E FC94  HOTV      JMP   HOTST     HOT START ENTRY POINT
FC09 7E FD93  INEEEV    JMP   INEEE     INPUT CHARACTER ROUTINE
FC0C 7E FDFD  OUTEEV    JMP   OUTEEE    OUTPUT CHARACTER ROUTINE
FC0F 7E FB79  CRLFV     JMP   PCRLF     PRINT CR/LF
FC12 7E FD85  PDATAV    JMP   PDATA     PRINT A STRING
FC15 7E FDED  INCH8V    JMP   INCH8
FC18 7E FD49  INHEXV    JMP   INHEX
FC1B 7E FD24  BYTEV     JMP   BYTE
FC1E 7E FD16  BADDRV    JMP   BADDR
FC21 7E FD5E  OUT2HV    JMP   OUT2H
FC24 7E FD36  OUTHLV    JMP   OUTHL
FC27 7E FD3A  OUTHRV    JMP   OUTHR
FC2A 7E FD69  OUT2SV    JMP   OUT2HS
FC2D 7E FD67  OUT4SV    JMP   OUT4HS
FC30 7E FD6B  OUTSV     JMP   OUTS
```

Listing 2. FCROM jump table.

```
                * COLDSTART INITIALIZATION
FC33 8E D07F  COLDST LDS   #$D07F     SET STACK TO MONITOR AREA

                * INITIALIZE I/O PORTS
FC36 CE 8000          LDX   #$8000
FC39 86 03            LDA A #03          PORT 0 AND 1 ACIA
FC3B A7 00            STA A 0,X
FC3D A7 04            STA A 4,X
FC3F 86 0B            LDA A #11
FC41 A7 00            STA A 0,X
FC43 A7 04            STA A 4,X
FC45 BD FFA5          JSR   VINIT        INITIALIZE VIDEO

                * SEE IF OTHER ROMS REQUIRE COLD START INITIALIZATION
FC48 B6 E000          LDA A $E000        CHECK ROM-E0
FC4B 81 7E            CMP A #$7E         IS THERE A JUMP?
FC4D 26 03            BNE   WARMST       NO
FC4F BD E000          JSR   $E000        YES, GO TO IT
```

Listing 3. FCROM cold-start initialization.

dexed jump to the address given in the index register. This address is actually specified through RAM and can therefore be changed by user programs, even though the JMP instructions themselves are in ROM. The three addresses used are exactly compatible with SWTBUG:

IRQ is A000
SWI is A012
NMI is A006

FCROM Jump Table

FCROM contains routines that are subject to future change. To avoid having to change other software, all these are handled through a jump table (sometimes also called a "transfer vector") as shown in Listing 2. In particular, note that FC00 is the start location to which the computer jumps on a reset. This is called COLDV (cold-start vector), and it jumps to COLDST at FC33. Two other entry points are WARMV and HOTV, followed by vectors or pointers to all the MIKBUG-compatible routines.

Cold Start

Listing 3 shows what happens at a reset (or cold-start; a jump to E0D0, which is the MIKBUG/SWTBUG reset address, also winds up at this location).

First, the stack pointer is set to point to the monitor stack at D07F. Then, MP-S ACIAs on ports 0 and 1 are reset and then initialized, followed by a jump to the video board initialization routine. In the case of HUMBUG, this is exactly the same as Percom's suggested video driver initialization, and so there is no need to show it here. If you have this video board, you already have a listing of it; if you don't, then you don't need it and can replace it with initialization for another video board or skip it.

The last four lines of cold-start check to see whether there is another ROM at address E000. Since all HUMBUG ROMs start with a jump table, we check to see whether there is a 7E or JMP instruction at address E000. If not, we continue to WARMST. If there is a JMP, we execute a JSR to E000.

```
                * E0 ROM ENTRY VECTORS

E000 7E E1AF  CINITV JMP   CINIT      COLD START INITIALIZATION

E1AC 7E FC09  INEEEV JMP   INEEE      VECTOR TO FC ROM

                * CINIT - COLD START INITIALIZATION

E1AF 01       CINIT  NOP              NONE REQUIRED FOR THIS ROM
                * SEE IF OTHER ROMS REQUIRE INITIALIZATION
E1B0 B6 E400          LDA A $E400     CHECK NEXT ROM
E1B3 81 7E            CMP A #$7E      IS THERE A JUMP?
E1B5 27 08            BEQ   CHORE4
E1B7 B6 E800          LDA A $E800     CHECK THE ROM AFTER THAT
E1BA 81 7E            CMP A #$7E      IS THERE A JUMP?
E1BC 27 04            BEQ   CHORE8
E1BE 39               RTS             NO, RETURN TO FCROM
E1BF 7E E400  CHORE4 JMP   $E400      YES, GO INITIALIZE
E1C2 7E E800  CHORE8 JMP   $E800      YES, GO INITIALIZE SECOND ROM
```

Listing 4. E0ROM cold-start initialization.

Cold-Start of Other ROMs

As it turns out, E0ROM doesn't need any cold-start initialization. Unfortunately, the overhead involved with the expandability of HUMBUG requires that we go through some testing to check for a following ROM (see Listing 4). Here we see the JMP at location E000, which leads to CINIT. Since E0ROM has many MIKBUG-compatible jumps, a lot of its routines have to be squeezed between these jumps. In this case, the CINIT cold-start initialization is placed right after the INEEE vector at E1AC, which is also shown in Listing 4.

The NOP at CINIT shows where the initialization would go, if there was some. The following steps check for a JMP at the start of the next ROM at address E400 and jump to it if present. If not, then they check for a JMP at the start of a ROM at E800 and again jump to it if present. If neither is present, then there occurs an RTS, which returns back to FCROM's warm-start procedure.

These steps check for a JMP both at address E400 and at E800, so that if an additional ROM is installed at E800, but the one at E400 is pulled out, then the system will simply skip past the removed ROM. The purpose is to allow the monitor to function at least partially, even if some of its ROMs are pulled out. The only crucial ROMs are FCROM and E0ROM, although the system will work even with just FCROM.

Although E0ROM doesn't need cold-start initialization, E4ROM does. Its cold-start initialization is shown in Listing 5. Notice that E4ROM tries to differentiate between a reset or jump to the cold-start location E0D0, as opposed to a real cold-start right after the first power-on. The reason is because breakpoints have to be handled differently.

When you first turn on the power, the list of breakpoints maintained by HUMBUG has to be erased so that, if any new breakpoints are established, HUMBUG doesn't accidentally

```
                * E4 ROM ENTRY VECTORS

E400 7E E409  CINITV JMP   CINIT      COLD START INITIALIZATION
                * CINIT - COLD START INITIALIZATION

                * CHECK WHETHER THIS IS POWERUP OR RESET OF SYSTEM
E409 CE 1234  CINIT  LDX   #$1234     CHECK POWER-UP LOCATIONS
E40C BC D028         CPX   POWUP
E40F 26 05           BNE   PUP
E411 BC D02A         CPX   POWUP+2
E414 27 1C           BEQ   RESET

                * INITIAL POWER UP SEQUENCE
E416 CE E6BF  PUP    LDX   #BKRETN    INITIALIZE BREAKPOINT ISR ADDRESS
E419 FF A012         STX   SWIJMP
E41C CE 1234         LDX   #$1234
E41F FF D028         STX   POWUP
E422 FF D02A         STX   POWUP+2    INITIALIZE POWUP FLAGS
E425 CE D036         LDX   #BKTAB
E428 86 FF           LDA A #$FF
E42A C6 0C           LDA B #12
E42C A7 00    BKERAS STA A 0,X        ERASE BREAKPOINT TABLE
E42E 08              INX
E42F 5A              DEC B
E430 26 FA           BNE   BKERAS     REPEAT IF NOT FINISHED

                * SEE IF OTHER ROMS REQUIRE INITIALIZATION
E432 B6 E800  RESET  LDA A $E800      CHECK NEXT ROM
E435 81 7E           CMP A #$7E       IS THERE A JUMP?
E437 27 08           BEQ   CHORE4
E439 B6 EC00         LDA A $EC00      CHECK THE ROM AFTER THAT
E43C 81 7E           CMP A #$7E       IS THERE A JUMP?
E43E 27 04           BEQ   CHORE8
E440 39              RTS              NO, RETURN TO FCROM
E441 7E E800  CHORE4 JMP   $E800      YES, GO INITIALIZE
E444 7E EC00  CHORE8 JMP   $EC00      YES, GO INITIALIZE SECOND ROM
```

Listing 5. E4ROM cold-start initialization.

clobber a program by restoring what it thinks is a prior breakpoint.

On the other hand, when you press the reset button or make a jump to the cold-start location E0D0 (or FC00), you don't want to erase the breakpoint table because doing so would make you lose track of locations that have been replaced by a break. So we need a way of telling the difference between the two kinds of resets.

For this reason, four locations in monitor RAM, called POWUP, and located at D028 through D02B, are used as a flag. When you first turn on the power, these locations will contain some random numbers. CINIT in E4ROM (Listing 5) checks the contents of these locations. If the contents are 12, 34, 12 and 34, respectively, then the program assumes that this is not a real cold-start, and so a jump is made to RESET. But at the first cold-start, these locations will be random and will therefore not contain this particular combination. (The chance of their just accidentally holding this number at power-up is about 1 in 4 billion!)

In that case, the routine at PUP will be performed. This initializes the address for an SWI to the return address BKRETN used for breaks, places the 12-34-12-34 combination in POWUP and erases the breakpoint table BKTAB. Then it goes to RESET. (Once POWUP is set to 12-34-12-34, all subsequent resets will skip this segment.)

The final part of the cold-start procedure again checks whether there are other ROMs, this time at E800 and EC00, and jumps to them if present. Otherwise, an RTS brings us back to FCROM, which will continue with the warm-start initialization. Remember that FCROM went to E0ROM with a JSR. Each ROM then continued to the next ROM with a plain JMP, so that an RTS will bring us all the way back to the first JSR in FCROM.

Next month we'll conclude the listing of this "Monitor to End All Monitors."■