Multimachine Games

Ken Wasserman and Tim Stryker
Mach 2 Software
96 Hammersmith Apts
Danbury CT 06810

There you are, staring into a poor dumb tube, spending hours trying to wheedle, cajole, flatter and coax your machine into coughing up a few more points, or maybe into reluctantly admitting every now and then: "YOU WIN!!!(bell)(bell)!!!" How much satisfaction is there in that, really? How much challenge? So you beat the computer. So what? So the computer beat you. Who cares? Do you ever long for a scenario something like the following?. . .

Tonight will be the final, deciding match of the battle series—the winner will have won the regional computer-club title and will be eligible for the national playoffs next month in San Diego. As you and your worthy opponent, both dressed in black, enter the room, a hush falls over the gathered assembly. You approach your respective consoles, and, at a prearranged signal from the presiding judge, the game begins.

The screen before you contains a wealth of information about the status and positioning of your various forces. You have two "windows" onto the field of play, one centered on your base, the other on your current tank. You see no sign of your opponent or his base in either window, for the field of play is very large: you know that he is out there somewhere, but, as the game begins, you have no idea where.

As you begin to move your tank out of your base, you find that it stays centered in its own window, thereby making previously unseen portions of the field visible to you, while, from the point of view of your base (which is immobile) your tank appears to move away from window center until shortly it disappears off the edge. Quickly reconnoitering your base perimeter, you begin to lay down mines to protect it from invasion. (These mines are visible to you but not to your opponent, to whom a square filled with one of your mines looks just like a stretch of virgin grassland.) As you do this, the steady clickety-click you hear from your opponent's keyboard tells you that he is not exactly idle either—he is probably mining the area around his base.

Cassettes containing Flash Attack *for the PET, at $15 each, and kits containing all the hardware needed to run* Flash Attack *and other games on the PET, including CB2 sound, at $15 each, are available from: Mach 2 Software, 96 Hammersmith, Danbury CT 06810.*

Quickly reconnoitering your base perimeter, you begin to lay down mines to protect it from invasion.

Or perhaps his base is well protected by mountain ranges, and he is now already actively seeking yours? Or maybe he has decided on the decoy ploy, and is building and mining an entirely false base to confuse you? You have no way of knowing!

Running out of mines, you frantically return to your base to restock, then rush out again to complete the mining operation. Suddenly you hear the sound of a mine exploding. Has your opponent run across your mine field already? Or did he, in his own haste, run afoul of one of his own mines? Thankful you had the foresight to make your mine fields orderly, you investigate: one of them is missing! Your opponent's tank is now badly damaged, but there are still four more where that one came from, and, more important, he now has some idea as to where your base is.

Out of mines again, and unwilling to return to base to restock, you are unable to patch the breach—instead, you take off after the intruder, and suddenly—there he is! His tank appears within your tank window! You fire—and miss—he maneuvers, fires—and hits you!

Your tank goes into condition yellow—you maneuver, fire—and miss—fire again—a hit! His tank, which was in condition red from having hit the mine, is completely destroyed, but you know that the second of his supply of five tanks has now been made available to him back at his base, wherever that is. Quickly slipping into a nearby forest to survey the area, you suddenly run across what can only be his second tank!

You reason as follows: in order for his second tank to have gotten back to this area as fast as it did, his base *must be nearby. Accordingly, you ignore the fact that his tank begins firing at you, opting instead to try to catch a glimpse of his base in your tank window before your tank is destroyed.*

You maneuver—are hit!—your tank is now in condition red, and you find it difficult to move properly—nevertheless you forge ahead—there is his base! You move again, and hit a mine—your tank is destroyed! However, remembering the coordinates your tank was at when you saw his base, you make a lightning mental conversion from rectangular to polar coordinates, and, shouting insults across the room to distract your opponent's attention, you swiftly key the polar data into your angle and range registers and fire off an intercontinental ballistic missile from your base. A high, falling whistle is heard, followed by a colossal explosion.

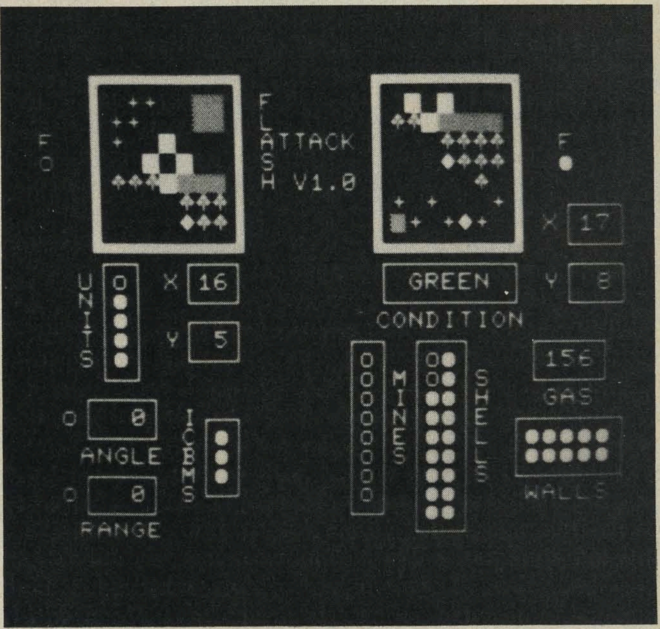
A deathly quiet ensues: your condition display glows with the word "SUPREME," while on your opponent's screen you know the condition to be "DEFUNCT." You have triumphed in the first game of tonight's seven-game match—as you glance across to see the look of fierce determination on the face of your opponent, you realize that the remaining games may not be won so easily. The judge, looking at both players, slowly raises his hands, and the second game begins. . . .

Creating a Game

The creation of such a game may not be as far beyond your capabilities as you might think: the above game, including all features mentioned, and more, has already been implemented for use on a pair of lowly 8 K-byte Commodore PET computers, under the name of *Flash Attack*. (See photos 1a and 1b.)

A pair of 16 K-byte PETs, TRS-80s, or Apple IIs should allow the development of even more outrageous games of this general type, perhaps involving quicksand pools, laser weaponry, or aerial reconnaissance, to name a few possibilities. The game could even conceivably be generalized to include more than two players, leading to situations in which teamwork and treachery could become determining factors in a game's outcome.

(1a)



(1b)

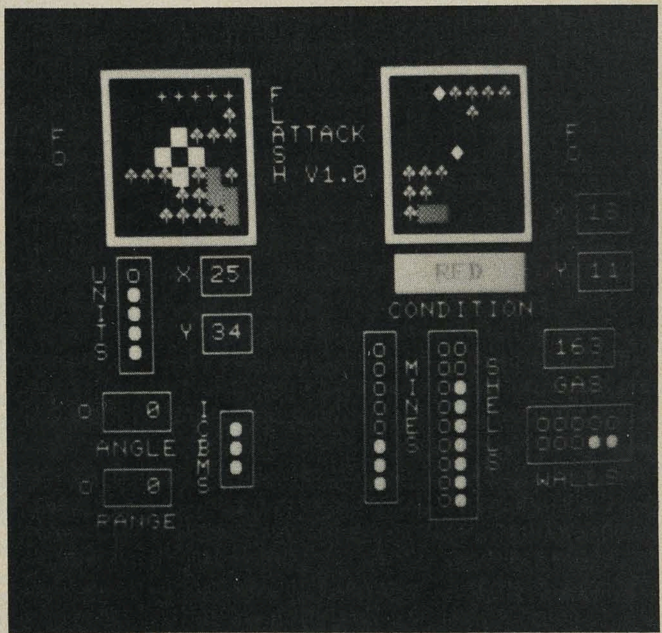
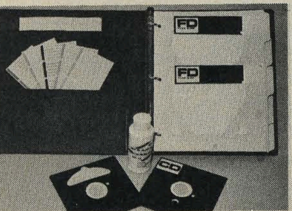


Photo 1: *A typical game of* Flash Attack *fully underway. The photos 1a and 1b show the display screens seen by each of the two players. The two rectangular "windows" seen on each screen represent a limited view of each player's base and the view from his active tank. By presenting only incomplete information to each player, the skill necessary (along with the corresponding sense of accomplishment) is increased.*

FLOPPY DISK FACTORY DIRECT FLOPPY DISK FACTORY DIRECT FLOPPY DISK FACTORY DIRECT FLOPPY DISK FACTORY DIRECT FLOPPY DISK FACTORY DIRECT FLOPPY DISK FACTORY DIRECT FLOPPY DISK FACTOR



- Buy factory direct from the manufacturer of floppy disks sold exclusively under other well-known brand names.
- Save distribution costs on reversible 5.25" floppy disks, which are 100% certified for 40 track, dual side, double density and all other standard applications.
- **50%** savings on 5.25" floppy disks with new factory-direct 2.2 megabyte capacity **"Diskit"**.
- Factory-direct **"Diskit"**, includes twelve floppy disks and accessories, plus a head cleaner disk and solution packaged in a bookcase binder for easy access, handling and storage. **"Diskit"** complete for the introductory price of only $49.50 plus $2.50 freight and handling. Normal price $59.50. **Order now and save $10.00** (CA residents add 6½% sales tax).
- Available in hard and soft sector configurations; when ordering indicate number of sectors — 1, 10 or 16.
- Send check or money order to:

Factory Direct
P.O. Box 60759
Sunnyvale, CA 94088
For VISA or MasterCard call toll-free
(800) 824-7888
In Calif. (800) 852-7777
In Alaska & Hawaii (800) 824-7919
Ask for Operator 98

Circle 18 on inquiry card.

Circle 19 on inquiry card. →

The game hinges on the players' judicious use of incomplete information.

The basic factors that go into making a game like this interesting are threefold:

1. More than one human player is involved in the game. Rather than having the user compete against the machine, the machine is *utilized* to permit two or more people to compete with *each other* in ways that would be impossible without the aid of the machine.
2. Success in the game hinges on the players' judicious use of *incomplete information.* Although the game may, in fact, be entirely deterministic in the sense that each legal move a player proposes gets put into effect without the intervention of any randomizing influence, the fact that each player has only a limited notion as to what his opponents are up to lends a definite element of suspense and calculated risk-taking to the game.
3. The game is played in *real time:* one's options are constrained not so much by the rules of the game as by one's own fleetness of hand and mind (or lack thereof).

Many conventional board games, and virtually all conventional card games, embody factors 1 and 2. Many video pinball parlor games, such as Atari's *Pong* and *Tank*, embody factors 1 and 3, while most of the rest of the available microcomputer game software embodies either none of these factors (computer chess, backgammon, etc), factor 2 alone (*Star Trek*, *Adventure*, etc), or, in exceptional cases, factors 2 and 3 together (real-time *Star Trek*, etc).

It is interesting to note that, of all the major league sports, the one that embodies all three of these factors most fully is football—this may be the reason why the sport is so overwhelmingly popular.

Bringing all three of these factors together in a single computer game virtually *requires* that more than a single console be used. Since, to most of us, a requirement for multiple consoles is equivalent to a requirement for multiple machines, the issue that will be addressed here is: what is needed in the way of hardware and software to support the implementation of multimachine games?

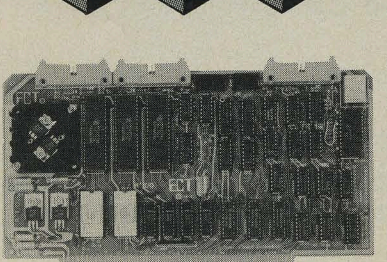
Two-Machine Games

In the case of two-machine games, the answer turns out to be surprisingly simple and inexpensive. Most microcomputers come already supplied with a general-purpose, 8-bit, parallel I/O (input/output) port poking out the back someplace. For those that do not, an add-on port of this type can generally be purchased at nominal expense. As in the PET, the port should ideally have the property that, even though configured for output, it will still return a correct reading of the states of the pins involved when a "read" operation is performed on it.

Also, as with the PET, the port should represent the *high* state upon output by means of a passive pull-up resistor. Ports not satisfying these conditions may still be

ECT™

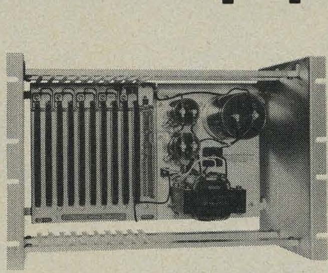
Building Blocks for Microcomputer Systems, Dedicated Controllers and Test Equipment.



R²I/O S-100 ROM, RAM & I/O BOARD

ECT's R²I/O is an S-100 Bus I/O Board with 3 Serial I/O Ports (UART's), 1 Parallel I/O Port, 4 Status Ports, 2K of ROM with the 8080 Apple Monitor Program and 2K of Static RAM.

$295.00



RM-10 S-100 RACK MOUNT CARD CAGE

ECT's RM-10 is a rack mount 10 slot Card Cage with Power Supply, consisting of an ECT-100 rack mount Card Cage (19"W x 12.25"H x 8"D), the MB-10 Mother Board (with ground plane and termination) all 10 connectors and guides and the PS-15A Power Supply (15A @ 8V, 1.5A @ ± 16V).

$295.00

Specializing in Quality Microcomputer Hardware
Industrial • Educational • Small Business • Personal
Card Cages, Power Supplies, Mainframes, CPU's, Memory, I/O, OEM Variations

ECT™ **ELECTRONIC CONTROL TECHNOLOGY** (201) 686-8080
763 Ramsey Ave., Hillside, NJ 07205

Circle 20 on inquiry card.

Listing 1: *Listing of the program used to test the cable described in figure 1.*

```
100 REM*** PROGRAM TO TEST INTER-
110 REM***    MACHINE COMMUNICATIONS
120 REM***
200 GOSUB 10000
210 IF PEEK(59471) AND 16 THEN 260
220 GET S$
230 IF S$ = "" THEN S$ = CHR$(0)
240 GOSUB 10200
250 IF S$ = "●" THEN 999
260 GOSUB 10400
270 PRINT R$;
280 IF R$ <> "●" THEN 220
999 END
10000 REM***
10010 REM*** ROUTINE TO INITIALIZE PORT
10020 REM*** INPUTS: NONE
10030 REM*** OUTPUTS: NONE
10040 REM***
10050 POKE 59471,255
10060 POKE 59459,255
10070 RETURN
10200 REM***
10210 REM*** ROUTINE TO SEND BYTE
10220 REM*** INPUT: S$ = BYTE TO BE
10230 REM***              SENT
10240 REM*** OUTPUTS: NONE
10250 REM***
10260 HN = INT(ASC(S$)/16)
10270 LN = ASC(S$)-HN*16
10280 POKE 59471,LN+128+32
10290 IF PEEK(59471) AND 128 THEN 10290
10300 POKE 59471,HN+128+64
10310 IF PEEK(59471) AND 128 THEN 10330
10320 GOTO 10310
10330 POKE 59471,255
10340 RETURN
10400 REM***
10410 REM*** ROUTINE TO RECEIVE BYTE
10420 REM*** INPUTS: NONE
10430 REM*** OUTPUT: R$ = BYTE RECEIVED
10440 REM***
10450 IF PEEK(59471) AND 64 THEN 10450
10460 LN = PEEK(59471) AND 15
10470 POKE 59471,127
10480 IF PEEK(59471) AND 32 THEN 10480
10490 HN = PEEK(59471) AND 15
10500 POKE 59471,255
10510 R$ = CHR$(HN*16+LN)
10520 RETURN
```

used as long as there is provision made within them for individually programming each bit position to be either input or output (examples of the use of such ports will not be given here).

What is needed, then, is an arrangement that will allow a byte at a time to be transferred from either machine to the other. Figure 1 gives the wiring diagram for the cable needed; as you can see, each bit position on each machine is simply directly connected to the corresponding bit position on the opposite machine. This is true for all bits except for the 2^4 bit, labeled ASYM, which is grounded on one machine and left floating *high* on the other. The whole package, including connectors, should cost less than $5.

Listing 1 contains a program designed to test the cable. It is designed for use on a pair of PETs, but, with minor modifications, it should be capable of supporting any pair of machines with ports satisfying the conditions discussed above. With the cable in place, and with both machines running this program, what should happen is that any keys hit on either machine should be displayed on the screen of the other. Type a shift-Q (*not* the STOP key) to exit the program and return to BASIC.

The three utility routines of interest here start at lines 10000, 10200, and 10400, respectively. The routine at line 10000 simply initializes the port: location 59471 is the PET's User Port I/O data register, while 59459 is the register used to configure the data pins for input and output. The POKE in line 10060 configures all eight pins as output.

The SEND routine at line 10200 may be called whenever it is desired to send a byte to the opposite machine. However, the opposite machine must call its own RECEIVE routine, at line 10400, in order for the transfer to take place. There is a potential pitfall here: if, when writing your own code to use these routines, you create a situation in which both machines are trying to send a byte to the other at the same time, or if both machines try to receive a byte from the other at the same time, both will "hang."

The programs running on the two machines must be set up in such a way that whenever one of them decides to send a byte, the other realizes this and sets up to receive it. Given this fact, the purpose of the ASYM bit in figure 1 becomes evident: it guarantees that start-up problems will not arise when running identical copies of a single program in both machines. Consider yourself in the position of the program in listing 1 as you begin running; eventually you would reach the point where you would like to start up a dialogue with the other machine.

Question: should you send a byte to the other machine first, or receive one? You and the other machine had better come to complementary conclusions as to which to do first. Solution: you use the setting of the ASYM bit to decide. This is exactly what happens in line 210 in the listing. If, upon reading the port contents, you find that the 2^4 bit is *high*, you receive first; otherwise you send first. From that point on, in this example, you simply alternate sending and receiving, and everything is fine.

Let's take a closer look at what is actually involved in transferring a byte using this scheme. The nine lines shown in figure 1 can be broken down into four groups:

- GND. This is a signal ground, which must be present in

Circle 22 on inquiry card. ⟶

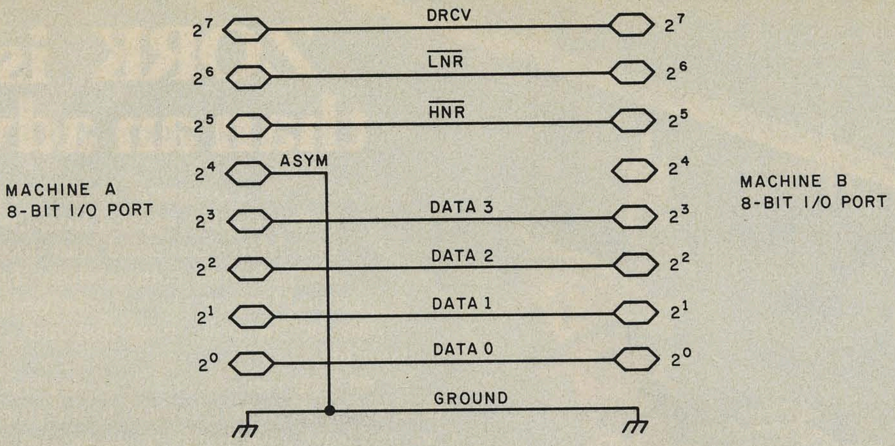


Figure 1: *The cable arrangement needed for connecting two PETs in game-playing configuration. Each machine runs the same program, and exchanges relevant information, one byte at a time, with the opponent's computer. The bit labeled 2^4 determines the initial state of each machine and, thus, whether it first transmits or receives.*

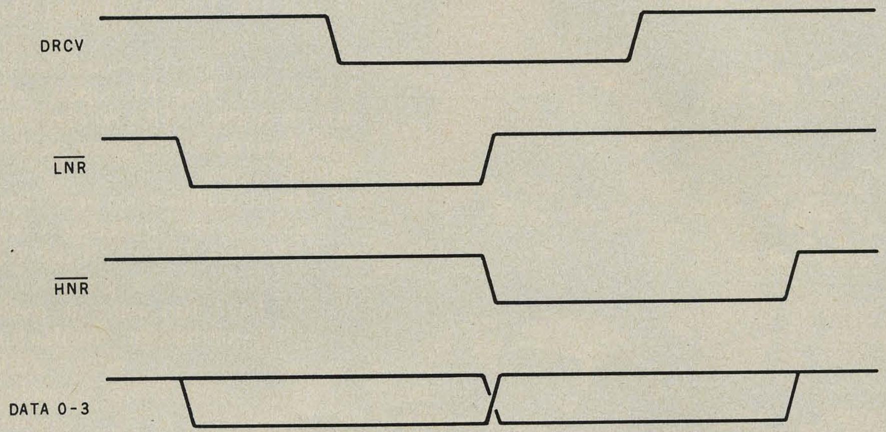


Figure 2: *Timing diagram for information transfer using the cable scheme of figure 1. The transmitting computer puts information on the DATA lines, low-order nybble first, and brings the LNR line low. The receiving computer brings the DRCV line low when the information has been accepted. The process is repeated for the high-order nybble, but HNR is used to indicate the presence of new data. When DRCV is brought high, the transmitter and receiver functions reverse.*

order for the two machines to have a common reference voltage.

- DATA 0 thru 3. These lines, which are controlled by the sender, carry the actual data being transferred, a nybble at a time (a nybble is half of a byte, or 4 bits).
- ASYM. This has already been discussed.
- DRCV, $\overline{\text{LNR}}$, and $\overline{\text{HNR}}$ (data received, low-order nybble ready, and high-order nybble ready). These are the so-called "handshake" lines. $\overline{\text{LNR}}$, which is a signal from the sender to the receiver, is brought *low* by the sender to indicate to the receiver that the low-order nybble of the byte being sent is now ready to be read off of the DATA lines. $\overline{\text{HNR}}$, also a signal from the sender to the receiver, is brought *low* by the sender to indicate to the receiver that the high-order nybble of the byte being sent is now ready to be read off the DATA lines.

DRCV, which is a signal from the receiver to the sender, is brought *low* by the receiver once he has read the low-order nybble off of the DATA lines, to indicate to the sender that he is ready for the high-order nybble; DRCV is then brought *high* again by the receiver once he has read the high-order nybble off of the DATA lines, to indicate to the sender that the high-order nybble has been received and that, as far as the receiver is concerned, the transaction is complete.

Figure 2 shows a timing diagram of the whole operation. Essentially, what happens is this:

The sender puts the low-order nybble on the DATA lines, and (by bringing $\overline{\text{LNR}}$ *low*) says, "Here is the low-order nybble." The receiver reads in the low-order nybble, and (by bringing DRCV *low*) says, "I've got it." The sender then puts the high-order nybble on the DATA lines, and (by bringing $\overline{\text{LNR}}$ *high* and $\overline{\text{HNR}}$ *low*) says, "Here is the high-order nybble." The receiver reads in the high-order nybble, combines it with the low-order one to make a complete byte, and (by bringing DRCV *high* again) says, "All set. Goodbye." The sender must then return all lines to the *high* state before returning to his caller.

All lines are left in the *high* state except when actually in use so that if one machine tries to send or receive while the other is off doing something else, the first machine will simply wait until the other is ready before proceeding with the transfer.

The only modifications necessary for this scheme (to handle ports lacking the previously discussed properties) would be: to have code at the beginning of the RECEIVE routine which configured the DRCV line for output and the remaining lines for input; to have code at the beginning of the SEND routine that configured the DRCV line for input and the remaining lines for output; and to have code at the ends of both routines for reconfiguring all lines as input. The port initialization routine would also have to be changed to initially configure all lines for input so that the ASYM bit could be sensed properly.

Although code resembling that shown in listing 1 works, it executes excruciatingly slowly under most current implementations of BASIC. Anyone considering writing a real-time game using these routines would be well advised to rewrite, at a minimum, the SEND and RECEIVE routines in machine language. Listing 2 shows a program, tailored for the PET, which is functionally identical to the one in listing 1: the difference is that in listing 2 all three utility routines have been implemented in machine code.

The subroutine at 10000 now sets up the machine code in the PET's "tape-2 buffer"—the SYS to 909 in line 200 is what actually initializes the port. The USR function is invoked with a negative argument (as in line 240) to cause the machine to execute the RECEIVE software . . . the value returned by USR is that of the byte received.

When the argument to the USR function is nonnegative (as in line 230), its value is turned over to the SEND software for transferrence to the other machine . . . under these conditions the value returned by USR is garbage. The ASYM bit must still be checked from BASIC to determine whether to send first or receive first. (See line 210.)

Listing 2: *Functionally the same as listing 1, this program is tailored for the PET computer and has several utility routines implemented in machine code.*

```
100 REM*** PROGRAM TO TEST INTER-
110 REM***    MACHINE COMMUNICATIONS
120 REM***    USING MACHINE LANGUAGE
130 REM***    AND OTHER EFFICIENT
140 REM***    PROGRAMMING TECHNIQUES
150 REM***
200 GOSUB10000:SYS909
210 IFPEEK(59471)AND16THEN240
220 GETS$:S=0:IFLEN(S$)>0THENS=ASC(S$)
230 U=USR(S):IFS$="▤"THEN999
240 R$=CHR$(USR(-1)):PRINTR$;
250 IFR$<>"▤"THEN220
999 END
10000 REM*** THIS ROUTINE SETS UP THE
10010 REM***    FOLLOWING FACILITIES
10020 REM***    IN MACHINE LANGUAGE:
10030 REM***
10040 REM***    SYS909 ...INITS PORT
10050 REM***
10060 REM***    U=USR(+N) ...SENDS N
10070 REM***
10080 REM***    U=USR(-1) ...RECEIVES U
10090 REM***
10100 FORI=826TO917:READX:POKEI,X:NEXT
10110 POKE1,58:POKE2,3
10120 IFPEEK(50003)=0THENRETURN
10130 POKE827,154:POKE830,97:POKE834,98
10140 POKE869,109:POKE882,98:POKE903,98
10150 RETURN
11000 DATA 32,167,208,166,179,208,32
11010 DATA 165,180,72,9,240,41,191,141
11020 DATA 79,232,104,74,74,74,74,9
11030 DATA 208,44,79,232,48,251,141,79
11040 DATA 232,44,79,232,16,251,48,44
11050 DATA 32,103,3,76,120,210,44,79
11060 DATA 232,112,251,173,79,232,41,15
11070 DATA 133,180,169,127,141,79,232
11080 DATA 169,32,44,79,232,208,251,173
11090 DATA 79,232,10,10,10,10,5,180,168
11100 DATA 234,234,169,0,162,255,142,79
11110 DATA 232,142,67,232,96
```

Putting It All Together

Just having the capability to transfer bytes back and forth between two machines does not guarantee success in writing multimachine games. We now need a general strategy for controlling the flow of information between the various machines in such a way that the moves made by each player are processed in a consistent manner by all machines involved. Among other things, the strategy used must ensure that all of the machines involved agree as to the order in which the various players' moves are to be processed. Only one such strategy, the key-oriented strategy, will be discussed here. Although many other approaches to the problem do exist, this one is particularly "clean" and therefore easily debugged; it is also reasonably efficient in both space and time.

The information transfers addressed by any general strategy of this kind fall into two groups: those that occur at initialization time and those that occur during the actual play of the game. The key-oriented strategy calls for all information pertinent to the initial state of the game, including information that may be kept secret from one or more players, to be made known to all machines at initialization time.

Then, during play, a continuous conversation is set up among the machines in which the only information changing hands consists of individual keystrokes generated by the players at their keyboards. If a player generates no keystroke to be sent on a given pass, a zero byte is sent out to the other machine(s) to indicate this fact. Every machine maintains the full status of every player but only displays the information its own player is supposed to see.

Listing 3 shows a program, Real-Time Two-Machine Hangman, designed to illustrate the use of the key-oriented strategy. To keep it short, such things as instructions, gruesome representations of gallows, and so on have been left out. The object of the game is not, as it is in normal Hangman, to guess your opponent's word within a set number of letter-guesses while he sits around telling you where your correct guesses fit in. Instead, both you and your opponent choose words that the other tries to

Circle 23 on inquiry card. ⟶

guess—whoever guesses the other's word first wins.

The program as shown is, of course, only capable of running on a pair of PETs. However, with suitable alteration of the SEND/RECEIVE software, it should be possible to run it on any pair of common microcomputers possessing the cabling arrangement described above.

Game Time

To play the *Hangman* game, attach the cable, type the program in, and RUN it on both machines. You and your

Listing 3: Real-time Two-Machine Hangman *in which you attempt to guess your opponent's chosen word first.*

```
10 REM*** REAL-TIME 2-MACHINE HANGMAN
20 REM***
30 REM***  W$... THE TARGET WORDS
40 REM***  F$... LETTERS FOUND SO FAR
50 REM***  T$... LETTERS TRIED SO FAR
60 REM***
90 DIM W$(2),F$(2),T$(2)
100 GOSUB 10000 : SYS 909
110 PRINT "WHAT IS YOUR WORD";
120 INPUT W$(1)
130 IF PEEK(59471) AND 16 THEN 190
140 LS=USR(-1) : U=USR(LEN(W$(1)))
150 IF LS<>LEN(W$(1)) THEN 210
160 GOSUB 5100 : GOSUB 5000
170 GOSUB 5200 : P=2 : GOTO 280
190 U=USR(LEN(W$(1))) : LS=USR(-1)
200 IF LS=LEN(W$(1)) THEN 230
210 PRINT "WORDS ARE NOT SAME LENGTH"
220 GOTO 110
230 GOSUB 5000 : GOSUB 5100
240 GOSUB 5200 : P=1
250 REM***
255 REM*** MAIN PROCESSING LOOP
260 REM***
270 M$=CHR$(USR(-1)) : GOTO 300
280 GET M$ : M$=MID$(M$+CHR$(0),1,1)
290 U=USR(ASC(M$))
300 IF M$=CHR$(0) THEN 500
310 FOR I=1 TO LS
320 IF M$<>MID$(W$(P),I,1) THEN 360
350 F$(P)=MID$(F$(P),1,I-1)+M$+MID$(F$(
P),I+1,LS-I)
360 NEXT I : IF F$(P)=W$(P) THEN 1000
390 T$(P)=T$(P)+M$ : IF P=1 THEN 500
400 PRINT
410 PRINT "WORD SO FAR: ";F$(2)
420 PRINT "TRIED SO FAR: ";T$(2)
500 P=3-P : ON P GOTO 270,280
1000 REM***
1005 REM*** WE HAVE A WINNER
1010 REM***
1020 PRINT : IF P=1 THEN 1040
1030 PRINT "YOU WIN" : GOTO 1100
1040 PRINT "YOU LOSE"
1100 PRINT "THE MAGIC WORD WAS: ";W$(2)
1110 END
5000 REM*** ROUTINE TO SEND ENTIRE WORD
5005 REM***     TO OTHER MACHINE
5010 REM***
5020 FOR I=1 TO LS
5030 U=USR(ASC(MID$(W$(1),I,1)))
5040 NEXT I : RETURN
5100 REM***
5105 REM*** ROUTINE TO RECEIVE ENTIRE
5110 REM***     WORD FROM OTHER MACHINE
5115 REM***
5120 FOR I=1 TO LS
5130 W$(2)=W$(2)+CHR$(USR(-1))
5140 NEXT I : RETURN
5200 REM***
5205 REM*** ROUTINE TO INITIALIZE BOTH
5210 REM***     F$ ENTRIES TO ALL DASHES
5215 REM***
5220 FOR I=1 TO LS
5230 F$(1)=F$(1)+"-" : F$(2)=F$(2)+"-"
5240 NEXT I : RETURN
10000 REM*** THIS ROUTINE SETS UP THE
10010 REM***     FOLLOWING FACILITIES
10020 REM***     IN MACHINE LANGUAGE:
10030 REM***
10040 REM***     SYS909 ...INITS PORT
10050 REM***
10060 REM***     U=USR(+N) ...SENDS N
10070 REM***
10080 REM***     U=USR(-1) ...RECEIVES U
10090 REM***
10100 FORI=826TO917:READX:POKEI,X:NEXT
10110 POKE1,58:POKE2,3
10120 IFPEEK(50003)=0THENRETURN
10130 POKE827,154:POKE830,97:POKE834,98
10140 POKE869,109:POKE882,98:POKE903,98
10150 RETURN
11000 DATA 32,167,208,166,179,208,32
11010 DATA 165,180,72,9,240,41,191,141
11020 DATA 79,232,104,74,74,74,74,9
11030 DATA 208,44,79,232,48,251,141,79
11040 DATA 232,44,79,232,16,251,48,44
11050 DATA 32,103,3,76,120,210,44,79
11060 DATA 232,112,251,173,79,232,41,15
11070 DATA 133,180,169,127,141,79,232
11080 DATA 169,32,44,79,232,208,251,173
11090 DATA 79,232,10,10,10,10,5,180,168
11100 DATA 234,234,169,0,162,255,142,79
11110 DATA 232,142,67,232,96
```

opponent will each be asked to enter a word—if the words entered are of different lengths, the program prints an error message and reprompts both players for new words. Once the program has accepted the two words, any key you strike is taken to be a letter-guess directed at your opponent's word.

Each time you hit a key, your machine displays the results of your guess—that is, your target word so far, with dashes in the positions corresponding to letters not yet guessed, and a tabulation of the letters you have tried so far. The program automatically detects when one player has guessed every letter in his opponent's word, and declares the winner accordingly.

The initialization phase of listing 3 encompasses lines 10 thru 240 and all of the subroutines appearing from line 5000 on up. During this phase, the program POKEs the machine-language software into place, initializes the port to the other machine, and then (in line 110) prompts its own player for input and reads the reply into W$(1).

Then, using the ASYM bit as usual to determine whether to send first or receive first, it essentially exchanges word lengths with the other machine and checks to make sure that the two word lengths are equal. Once satisfied that they are, the program proceeds to exchange words with the other machine (using the subroutines at 5000 and 5100), placing the other player's word into W$(2). Both machines now know both players' words. Each machine has its own player's word in its own copy of W$(1) and the opposing player's word in its own copy of W$(2).

The Play Phase

At this point, the program is ready to enter the play phase, but first it must set the initial value of the *player select* variable P to either 1 or 2, depending on the setting of the ASYM bit. The reason for this is that the section of code from line 300 to line 500 is used to process proposed letters, or moves originating from *both* players—this is the essence of the key-oriented strategy. The variable P, which flips back and forth during play between 1 and 2 via the statement "P=3−P" in line 500, is used on each pass to determine whether to attempt to get a keystroke from one's own keyboard (which is what the GET statement in line 280 does) or to receive from the other machine the result of its attempt to get a keystroke from its own keyboard (which is what the assignment in line 270 does).

The value of P is also used in the main processing loop as the index into each of the two-element arrays W$, F$, and T$, to ensure that the proper player's status is updated as a result of the processing of the keystroke. The net implication is that P must be initialized to 1 on one machine and to 2 on the other so that the play phase will begin correctly.

During the play phase, then, the program simply circulates in the main processing loop shown, alternating the value of P back and forth between 1 and 2 on each pass. When P is 2, the machine's own keyboard is interrogated, the resulting keystroke (or a zero if the resulting keystroke was null) is sent off to the other machine, and the keystroke is processed by examining W$(2) for occurrences of it. F$(2) and T$(2) are updated accordingly and, in lines 410 and 420, are printed out.

When P is 1, the keystroke to be processed comes from the other machine (in order for this to happen the other machine's copy of P will at this point be equal to 2). The keystroke is processed by examining W$(1) for occurrences of it, and F$(1) and T$(1) are updated but not printed out, since they are of interest only to the player on the other machine.

Checking for the end-of-game is thus very simple: as soon as F$(P) becomes equal to W$(P), the game is over, and the value of P for which this was the case can be used (as it is in line 1020) to determine who won.

This is how a typical real-time two-machine game involving incomplete information is implemented. Other good candidates for implementation in this manner would be *Star Trek*, *Kriegspiel* (a version of chess in which neither player is ever entirely sure just where his opponent's pieces are located), and *Stratego*. You can easily design entirely new Adventure games, a submarine battle for example, using the basic approaches given here. The possibilities are certainly more exciting and creative than playing *Battleship* with pencil and graph paper.■

MARK GORDON COMPUTERS

DIVISION OF MARK GORDON ASSOCIATES, INC.

P.O. BOX 77, CHARLESTOWN, MASSACHUSETTS 02129
(617) 491-7505

SD SYSTEMS COMPUTER KITS

★ EXPANDORAM I (No RAMS)	169.00
★ VERSAFLOPPY CONTROLLER I	189.00
★ SBC-100 Single Board Kit	239.00
★ Z80 Starter	269.00

OTHER SPECIALS

★ 16K Memory Kit	49.00
★ CAT Modem	151.00
★ Leedex Monitor	109.00
★ 16K Model III TRS-80	859.00*
★ Atari 800	779.00
★ Hazeltine 1410	699.00

To Order Call Toll-Free 1-800-343-5206

***TRS-80 is a Trademark of Tandy Corp.**

ORDERING INFORMATION

We accept Visa and Mastercharge. We will ship C.O.D. certified check or money order only. Massachusetts residents add 5 percent sales tax.

The Company cannot be liable for pictorial or typographical inaccuracies.

Need help? Call us.
Want to help?
Call us.

Red Cross is counting on you.

Circle 26 on inquiry card.

Circle 27 on inquiry card. ⟶