HC Journal™
Home Computing Journal

Welcome to: Volume 4

—Intelligent Software for Personal Computers in a Home Environment

PRODUCTION NOTE: This printed journal was produced in camera-ready format with a desktop publishing system employing an Apple Macintosh and LaserWriter system, and the following software: MacPaint, MacDraw, and MacWrite from Apple Computer, Inc; PageMaker from Aldus Corporation; Microsoft Word from Microsoft; plus Desktop Magic and LaserMagic from World Class Software.

Creative Staff:
Randy Thompson
Roger Wood
Walter Hego

Editorial Consultant:
Gary M. Kaplan

Home Computing Journal (HCJ) is a quarterly multi-media software subscription service containing ready-to-run productivity, education, entertainment, and utility programs on a floppy disk. The accompanying workbook contains the required support documentation plus additional technical notes and programming aids. For current single-copy and subscription pricing, please see last page. HCJ assumes no liability for errors in articles, programs, or workbook material.

Each separate workbook selection and software contribution to this Volume and the Volume as a collective work is Copyright 1987 by Home Computing Journal. All rights reserved. Visual elements and design of all pages Copyright 1987 by Home Computing Journal. All rights reserved.

Home Computing Journal is the owner of all rights to the computer programs and software published in this printed workbook and on its companion disk. To allow for convenience in the use of this software by the purchaser, HCJ grants to such purchaser only, the Limited License to enter these programs into the purchaser's computer, place a disk-copy of this software on the purchaser's hard-disk drive, and/or make a reasonable number of back-up safety copies for the purchaser's personal use. Any other use, distribution, sale, or copying of this software is expressly prohibited and is in violation of this Limited License and the copyright laws.

All indicated trademarks (TM), unless otherwise specified, are the property of Home Computing Journal.

Individual volumes and anthologies are available. See last page and /or inquire for current pricing and catalog:
Home Computing Journal
P.O. Box 70248 • Eugene, OR 97401
Tel. (503) 342-4013

Grid-Grappler

For guidance in grappling with the gremlins of computer intelligence, take a gander at this gripping game!

Designing and writing a computer game that pits the computer against a human opponent can be quite challenging. On the lowest level, there's the job of just getting the computer to follow the rules, while assuring that the human adheres to the same rules. Then the real fun begins: creating a computer opponent that is a worthy adversary for an intelligent and experienced human player.

The Rules Of *Grid Grappler*

To demonstrate how to create such a program, we present the game, *Grid Grappler*. This is a challenging game of strategy with a deceptively simple design: a grid with 25 (5X5), 49 (7X7), or 81 (9X9) squares which you and the computer take turns claiming. There are only two rules: (1) neither you nor the computer is allowed to take the middle square on the first move; and (2) once a square has been claimed, the other player cannot claim a square either horizontally or vertically adjacent to it.

The computer keeps track of the available squares. If you have at least one move left, it lets you take your turn. If you do not have a move, it keeps claiming squares until *it* runs out of moves. Similarly, if the computer has no moves left, it lets *you* take multiple turns until you run out. When there are no moves left for either of you, the round is over.

The player who has claimed the most squares is the winner of the round. The winner's point total is calculated as the difference between the number of squares claimed by each player. For example, if the winner claimed 10 squares, and the loser claimed 8, the winner would receive 2 points. The game is over when either player's points reach the number of squares on one side of the grid—e.g., if there are 7 squares on one side (49 total), then the first one to get 7 points wins.

At the end of each round, the computer figures the score and tells you who won that round. It then computes the running score for the game and checks to see if either player has won. If there is no winner, the computer informs us of the outcome, and asks if we want to play again. In addition to keeping track of the score of each game, a running score is maintained for all the games played since starting up the program. This allows you to see how you are doing against the computer over the course of several games.

Figure 1.
This chart demonstrates how different values in the GR array indicate the status of squares on the grid.

Figure 1

Number in GR(,) array	Control of Square	Available to
0	No one (initialized value)	Either player
1	Computer controls	Computer only
2	Human controls	Human only
3	Either or both	No one
88	No one (outside grid)	No one

These rules provide the complete specification for our program. They dictate that we must create an interactive graphic display for the grid, and an input routine for the human to select squares. Of course, the computer must have an internal method for keeping track of what moves have been made, and what moves are still available. Additionally, it requires some means of checking when one or both of the players runs out of legal moves; this naturally requires an end-of-game routine, with score-keeping and replay options. Although none of these tasks is trivial, our primary focus here is upon how the computer is programmed (with different levels of intelligence), to decide the best move at any point in the game. We'll discuss the user interface, graphics, and other elements only where they relate to how the computer selects its moves.

Tracking The Game

To understand how the computer makes decisions, you must first know how the program keeps track of the game board. It is stored as a two dimensional, numeric array **GR(,)** with 10 elements in each dimension. Each element stands for a square on the board—the first subscript identifying the X coordinate, the second the Y coordinate. For example, array element **GR(1,1)** represents the upper-left corner of the grid. The board is stored in elements with subscripts 1 through GS, where GS is a variable containing either 5, 7, or 9 depending upon the the Grid Size chosen by the player at the outset. The program places values in these variables depending upon the board position. Note that although the array elements with subscripts of **0** and GS+1 do not correspond to any square on the grid, these elements are used to help the computer decide its moves. More on this below.

In the following discussion, be aware of the important distinction between *claiming* a square (actually selecting the square so it will be filled with your color), and *controlling* a square (occupying a square either horizontally or vertically adjacent to it). Each time you claim a square, you automatically control the squares vertically and horizontally adjacent to it, and your opponent cannot claim those squares. Both players, however, may control the same square by each claiming squares adjacent to it. When this occurs, neither player is able to claim that square.

When a round begins, all active elements in the **GR(,)** array (that is, those between 1 and GS) are initialized to 0 (zero). When a square is claimed by either player, that player's score is incremented (scores are held in variable **C1** for the computer, **H1** for the human), and a 3 is placed in the corresponding element of the **GR** array—indicating that the square is no longer available for a move. In addition, the four squares horizontally and vertically adjacent to the square being claimed are marked to indicate who controls them (see Figure 1). If the initial value of an adjacent element of the **GR(,)** array is zero (no one yet controls the square), then the value of that element is incremented by 1 if it is the computer's move, or 2 if it is the human's move. If the opponent already controls the square in question, then its value is changed to a 3, denoting that the square is controlled by both players. Thus any square with a 3 in the corresponding element cannot be claimed because it is either already claimed by one of the players, or is controlled by both players. Similarly, any square that is controlled by the opponent cannot be claimed.

Look at Figure 2, a graphic representation of the **GR(,)** array for a game played on the 5X5 grid. The first dimension is labeled across the top (X), and the second dimension is labeled down the side (Y). The central 5X5 area is identical to the board as it is displayed on the screen. The outside row, however, is *not* part of the playing area and is, therefore, not displayed on the screen. This part of the **GR(,)** array is there to aid the computer in finding the best move.

In the position depicted in Figure 1, the human has selected the square directly above the middle square, and the computer has selected the square just to the left of the middle. The numbers in the squares show how the program has updated the **GR(,)** array. Both of the elements corresponding to the claimed squares now contain 3. Likewise the squares that *both* players control—**GR(3,3)** and **GR(2,2)**—contain 3s. The two squares that are controlled solely by the human—**GR(3,1)** and **GR(4,2)** contain 2s, and those controlled exclusively by the computer—**GR(1,3)** and **GR(2,4)**—contain 1s. All other squares in the grid still hold their initialized value of 0. The squares outside the grid were arbitrarily given a value—88—outside the range of 0-3. As you will see later, this could be anything as long as it is not in the range of 0-3.

How The Computer Chooses A Move

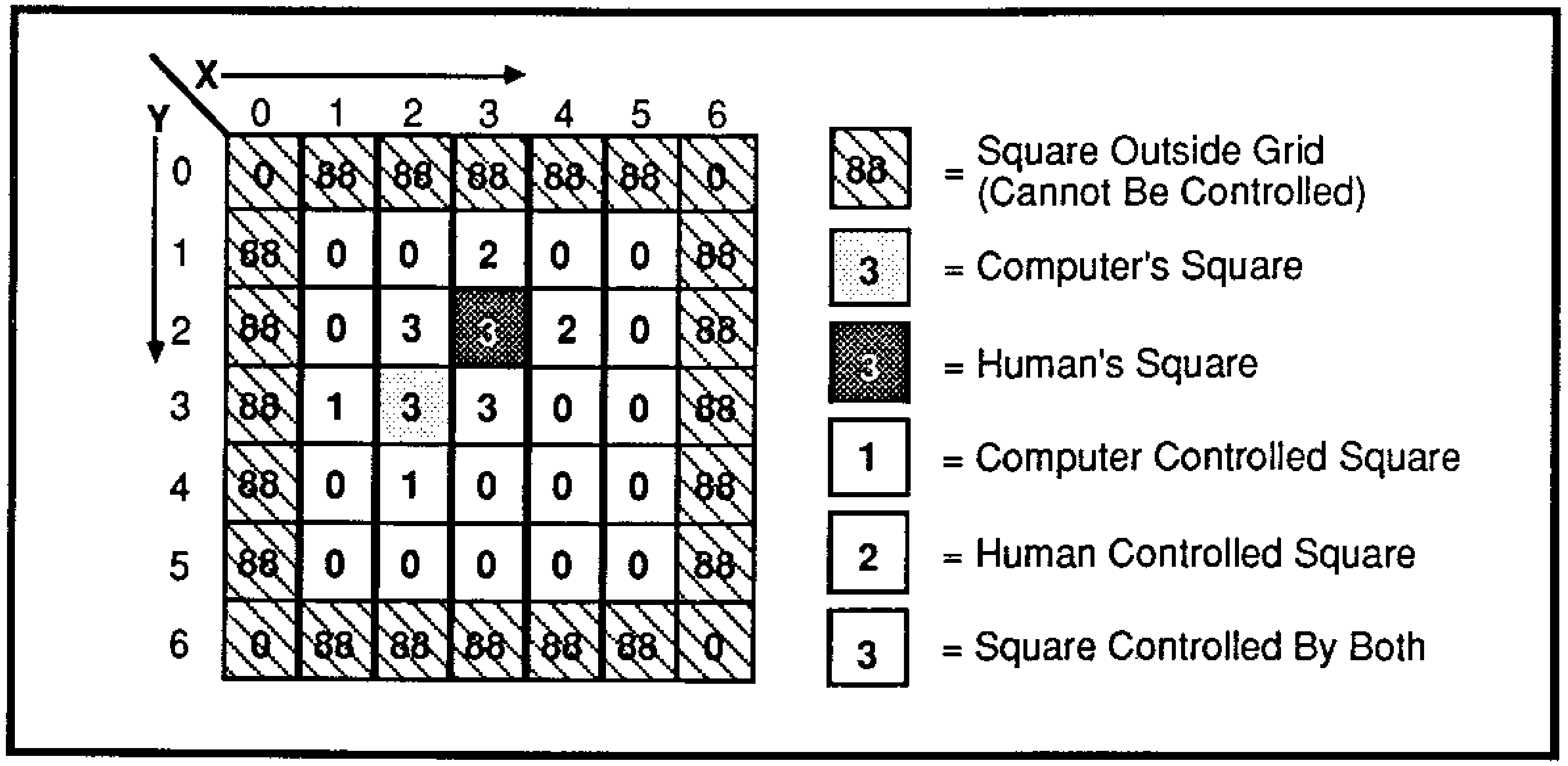
The computer makes its decision according to the algorithm we endow it with. The logic is straightforward: Upon checking the **GR(,)** array element for every square on the board, if the element contains a 3 or a 2, then the square is rejected as a possible move (because it is already occupied or the opponent controls it). If a square is available, the computer evaluates it in terms of the values of the four adjacent squares.

Let's consider for a moment what moves would be more offensively or defensively oriented: If the adjacent square contains a 0 (no one controls it), this move would be more of an offensive move. On the other hand, if the space contains a 2 (the human controls it), then moving next to it would tend to be more of a defensive move. Both of these moves would definitely advance our board position, and should therefore be given a positive weighting in our final decision.

But what if the adjacent square contains 3? To move in next to such a square would *not* change the status of that square at all. We must either already share control of it with our opponent, or occupy it ourselves. In short, moving into a square with a 3 adjacent to it does *not* help extend our control at all. This move should be discouraged.

What if the adjacent square contains a 1 indicating that we already control it? Although moving next to this square would not hurt our position, it would not be as good a move as one where we extend our control (if it contains a 0) or limit our opponent (if it contains a 2). This move, therefore, should be weighted somewhere between the good moves (next to a 0 or 2) and the bad moves (next to a 3).

Figure 2.
Here we see how the values stored in the GR(,) array correspond to the state of the playing grid.



Y \ X	0	1	2	3	4	5	6
0	0	88	88	88	88	88	0
1	88	0	0	2	0	0	88
2	88	0	3	3	2	0	88
3	88	1	3	3	0	0	88
4	88	0	1	0	0	0	88
5	88	0	0	0	0	0	88
6	0	88	88	88	88	88	0

88 = Square Outside Grid (Cannot Be Controlled)
3 = Computer's Square
3 = Human's Square
1 = Computer Controlled Square
2 = Human Controlled Square
3 = Square Controlled By Both

Figure 4.
This BASIC code in Listing 1 shows how the W( ) array variables are initialized to the correct weighting values. Then, in Listing 2, we use this weighting to calculate the relative strength of each possible move. Note that we employ pseudo line numbers; refer to the Program Legend to find the actual line numbers used for your computer.

Figure 4.

Program Listing Examples

Listing 1:

```
XX00 REM INITIALIZE PROGRAM
XX10 DIM W(2):W(0)=1.5:W(1)=2:W(2)=-2
```

Listing 2:

```
YY10 XY=GR(C-1,D):TS=TS-W(0)*(XY=0)-W(1)*(XY=2)-W(2)*(XY=3)
YY20 XY=GR(C+1,D):TS=TS-W(0)*(XY=0)-W(1)*(XY=2)-W(2)*(XY=3)
YY30 XY=GR(C,D-1):TS=TS-W(0)*(XY=0)-W(1)*(XY=2)-W(2)*(XY=3)
YY40 XY=GR(C,D+1):TS=TS-W(0)*(XY=0)-W(1)*(XY=2)-W(2)*(XY=3)
```

Program Legend

Pseudo Line #	APPLE	C-64	IBM	TI-99/4A
XX00	190	190	220	170
XX10	210	210	240	210-240
YY10	1150	1490	1180	1080-1090
YY20	1160	1500	1190	1100-1110
YY30	1170	1510	1200	1120-1130
YY40	1180	1520	1210	1140-1150

Figure 3.
This chart details the weighting values that we've assigned to the elements of the W( ) array. By changing these array values at the beginning of the program, you will alter the computer's playing strategy.

Figure 3.

Moving Next To	Weighting Given	Assigned Variable
0	1.5	W(0)
2	2	W(1)
3	−2	W(2)
1 or 88	0	None

This brings us to the elements on the outside edge—i.e., an adjacent element of the array without a corresponding square on the grid. (Remember, this element was arbitrarily given a value of 88.) Moving next to one of these is quite similar to moving next to a square we already control—it neither adds to nor detracts from our position in any appreciable way. We thus want this move to be given equal weighting to a move that is next to a square we already control (i.e., with a value of 1).

Turning our analysis of good and bad moves into a mathematical value is accomplished by first providing a weight to each of the four squares adjacent to a move, and then adding up the weight of the four squares to get a relative value for the move. To illustrate, let's evaluate some possible moves the computer might make given the position depicted in Figure 2. We will give these moves the following weight—thus establishing a slight edge for a defensive move:

(1) Moving next to a 0 is a good offensive move, and we give it a value of 1.5.

(2) Moving next to a 2 is a good defensive move, and we give it a value of 2.

(3) Moving next to a 3 is an inferior move, so we give it a (negative) value of −2.

(4) Moving next to a 1 or an 88 is a neutral move, so we give it a value of zero (see Figure 3).

Although these values would be assigned more or less arbitrarily while developing the program, we are using the values that we finally settled upon after testing the algorithm in the actual game.

Let's now evaluate three possible moves—i.e., GR(4,1), GR(4,3), and GR(4,4).

Square GR(4,1) is a very defensive move, taking sole control of squares GR(3,1) and GR(4,2) away from the human. By our weighting algorithm, this square is given the value of 4 [2+2+0+0].

Special Note To Commodore 64 Users:

To best take advantage of the graphic power of the C-64, *Grid Grappler* was written using the *Programmers' BASIC Toolkit*—a special programming package from EPYX. This product allows for direct access to bit-map graphics within a more-or-less BASIC language environment. To allow you to execute this program, we've included three files on your HCJ Volume 4 disk: GRID, GRB, and C. If you wish to make a backup of the program, you must ensure that all *three* files are on the backup disk.The program C can be copied by simply **LOAD**ing and **SAVE**ing, but GRID and GRB must be copied using a utility like *HCJ Duplicator*. [See Home Computing Journal, Volume 3, for the *HCJ Duplicator* program and complete instructions on using it. This is an excellent utility for backing up your HCJ disks—Ed.]

The file GRID, is a program that should always be loaded using this command: **LOAD"GRID",8,1** because it must load into a part of memory different from the normal BASIC default area at 2049 ($801). When you enter this command with your HCJ Volume 4 disk in the drive, the computer automatically loads the file GRB, which wedges in the proper machine language routines, and then runs the program C, the actual *Grid Grappler* program.

This is the only way you can use the program if you do not have the *Programmers' BASIC Toolkit* disk. You can't, however, modify or even LIST the program when it is loaded this way. But, you can make the modifications described in this article, if you follow the following procedure:

1. Reset your C-64 system by turning the power off and on.
2. LOAD the C program using the **LOAD"C",8** BASIC command.
3. LIST only the lines you wish to change. There are certain lines in the program that cannot be LISTed (e.g., if you try to LIST line 200 you will get a SYNTAX ERROR? message). In addition, because the language redefines certain BASIC tokens, certain lines will contain nonsense BASIC commands which will be redefined when RUN under the *Programmers' BASIC Toolkit* environment. Therefore, do *not* try to change these lines.
4. Make any changes just as you would any BASIC program. For example, to give moves a heavier weighting, making the computer play even more defensively, LIST line 2100 and change the variable W(1) to equal 3. Then SAVE the program as C. Note: You must scratch the original program C either by using the SAVE with replace option (**SAVE"@Ø:C",8**) or through OPENing the command channel and scratching the C file (**OPEN15,8,15,"SØ:C":CLOSE 15**) before saving (**SAVE"C",8**).

Note: *The Programmers' BASIC Toolkit is a trademark of Epyx, Inc. Copyright © 1985 Epyx, Inc. The suggested retail value of the product is $19.95. To obtain the package, contact either your local dealer of C-64 software or write*

Softmail
2995 Woodside Road
Suite 400-383
Woodside, CA 94062
(408) 848-3042

Moving to square **GR(4,3)** will be a more offensive move because of the weight given to squares **GR(5,3)** and **GR(4,4)**. The negative value of moving next to square **GR(3,3)**, however, gives this move the value of 3 [1.5+1.5+2+(–2)].

Taking square **GR(4,4)** is an offensive move because it results in control of four uncontrolled squares—**GR(4,3)**, **GR(5,4)**, **GR(4,5)**, and **GR(3,4)**. According to our rules above, this square is given the value of 6 [1.5+1.5+1.5+1.5]. If you check out every square on the board using the rules we've defined for weighting moves, you will find that this is definitely the "best" move.

Turning The Idea Into Code

Given this very limited testing, our initial algorithm seems to produce a reasonable response to the position. Let's see how this idea was turned onto BASIC code. First, we DIMension a small array, **W( )**, to hold the various Weights. Although we have distinguished 4 different classes of squares, we only need 3 elements in our Weight array because one class (those with values of 1 and 88) are given a weight of zero. The **W(0)** variable will hold the weight for our offensive move, the **W(1)** variable the weight for the defensive move, and **W(2)** for our bad moves (see Figure 3). These are initialized in the code of Listing 1 of Figure 4.

(Note: The line numbers in Figure 4 are *pseudo* line numbers representing the actual ones used in the different versions of the program. See the Program Legend of Figure 4 for the actual line numbers that apply to your particular computer.)

Coding the Computer's Move

Having declared and initialized these variables, we will now investigate how the computer actually selects its move. Because the center square is considered the most advantageous position, the computer always claims it, if possible. If it cannot, either because it is the first move, the square is already occupied, or it is controlled by the opponent, the computer then goes ahead implementing the algorithm described above.

By employing two FOR-NEXT loops, every element in the **GR(,)** array with a corresponding square is checked. Specifically, the FOR-NEXT loop counters, **C** and **D**, go from 1 to GS, and the element **GR(C,D)** is evaluated in turn. If a square contains either a 2 or 3, the computer cannot occupy it, and so the next element is checked. If the computer can occupy a square, a temporary variable (TS) is initialized to zero and the code in Listing 2 (see Figure 4) is executed.

The lines in Listing 2 analyze the squares *adjacent to* the square in question, and set TS equal to relative strength of the move according to the rules set forth above. XY is set, in turn, to the value of each of the squares adjacent to the square represented by GR(C,D). After these four lines have been completed, the square's relative value is contained in the TS variable. As each square's TS value is obtained, it is compared with a variable SC. This is initially set to –100. If TS is greater than SC, then SC's value is changed to that of TS. After all squares are evaluated, SC will be equal to highest value of TS, and thus the best available move. If TS is discovered to be equal to SC, then the computer randomly chooses one of the two locations.

The only time SC remains at –100 is when all squares contain a 2 or 3; the computer, therefore, no longer has a move. Before this condition exists, however, as successively higher values of TS are found, the location of the *current* best move is saved by placing the values of C and D into the variables A and B, respectively. Therefore, GR(A,B) contains the location of the best move when the FOR-NEXT loops are complete.

Altering The Computer's Logic

We've made it easy for you to alter the computer's primary algorithm for determining its move. You only have to alter the values in the **W( )** array. For example, because the **W(0)** element is the weight of a more defensive move, increasing its value will tend to make the computer play more defensively. Similarly, raising the value of **W(1)**, will make the computer search for squares that no one controls, and it will thus tend to play a more offensive game. Needless to say, we have tried to create an all-around good player, so the weights of these variables in the programs have been determined through a good deal of trial and error, testing against a variety of opponents.

If you find that the computer is just too good, try changing the value of **W(2)** from negative to positive. This will make the computer consider certain very poor moves to be good ones.

Although altering the **W( )** array is the easiest way to modify the computer's game, you could try making more extensive changes. Instead of just looking at the horizontally and vertically adjacent squares, you could expand your search to look at squares *diagonally* adjacent to each square. This would be accomplished by writing code that alters the value of TS by indexing the **GR(,)** array with **(C–1, D–1), (C+1,D–1), (C–1,D+1), (C+1,D+1)** in addition to the squares already being checked. Furthermore, you could even look *two* squares away, for example **(C+2,D)**. But this change would require some range checking, or you could exceed the DIMed size of the **GR(,)** array.

So, there you have it—an algorithmic approach to providing a computer with intelligence. With some relatively minor alterations, this same approach can be employed in a variety of gaming environments. The next time you're trying to computerize a logic puzzle, remember that mathematically weighting various alternatives allows the computer to quickly draw conclusions about the relative value of its options. Come to think of it, if we humans weren't so influenced by our emotions, this is the way we *too* would be making "perfect" decisions. But—as Mr. Spock would observe—this "flaw" is what makes humans so interesting...

Key Variables

Grid Grappler

Variable	Function
GR(,)	Grid array
MD	Middle of Grid
C1	Number of squares claimed by computer
H1	Number of squares claimed by human
X	Currently selected horizontal position
Y	Currently selected vertical position
C,D	Loop counters for evaluating moves
XY	Temporary variable to hold GR(,) value of square currently being considered
SC	Mathematical value of best square thus far considered
TS	Mathematical value of square currently being considered

Control Capsule

Grid Grappler

Key	Function
Cursor Left	Move to left on grid
Cursor Right	Move to right on grid
Cursor Up	Move up on grid
Cursor Down	Move down on grid
Space Bar*	Select current square

* **TI-99/4A users:** The A key is used for selecting the current square on the TI-99/4A instead of the Space Bar.

IS-Base Construction Set

Design and build customized IS-Base files with this powerful database accessory.

There are basically two types of database users: There are those who like to dump all available information into one gigantic file, and there are those who like to place information in several small neatly segregated files. While both techniques have their merits, neither is necessarily superior.

Our *IS-Base* program presented in Volume 3 was designed to work well with *both* types of users: the "stuff-it-all-in-one-file" mentality, and the "a-file-for-every-type-of-information" mindset. For those who wish to *alter* their style of data storage, however, *IS-Base* is a bit less accommodating. It lacks the ability to break up a large file into several smaller files, or merge several smaller files into one large file. To fill this crucial information-access gap, we now present *IS-Base Construction Set.*

What Does The Program Do?

IS-Base Contruction Set performs two basic operations: (1) it builds new *IS-Base* files using selected information from pre-existing files; and (2) it merges selected information from one or more *IS-Base* files with other pre-existing files.

When you build a file, information is selectively pulled from one *IS-Base* file and stored as a *new* file on a separate disk. The original file is unaffected by this procedure. Through the use of the Build File operation, you can "filter" your *IS-Base* files, creating new, more specialized files of information.

When you merge files, information is selectively pulled from one *IS-Base* file and *combined* with another file on a separate disk. The original file is unaffected by this procedure. Through the use of the Merge File operation, you can "glue" several files together, creating one master file of information.

Search parameters are used to specify the information to be merged or built. (For more information on search parameters, refer to the original *IS-Base* documentation in Volume 3.) Only information that matches the search parameters is used in a build or merge operation.

Because a separate disk is required for each *IS-Base* file, this program requires disk swapping on single-drive computer systems. The number of disk swaps depends on the file size and memory capacity of the computer. This program can be set for either single or dual-drive operation.

The Menu

IS-Base Contruction Set offers the following menu options:

```
1) Change Search Parameters
2) Change Target Drive
3) View Source File
4) Build File
5) Merge File
6) Exit Program
```

1) Change Search Parameters

This option allows you to change the search parameters. The current search parameters are shown at the bottom of the screen, below the menu. When you first boot *IS-Base Construction Set,* the search parameters default to the asterisk (*) wild card. If you use the wild card search parameter, *all* information found in the original source file is built or merged into the new file. Any search parameter available in *IS-Base* can be used in *IS-Base Construction Set.*

Here's a review of the five search parameter formats. The <...> symbol represents the particular piece or pieces of information that you are looking for.

Format 1: `<...> IS <...>`

This format indicates a search for information on *both* sides of the IS command. This command is useful when searching for one specific *IS-Base* entry.

Format 2: `<...>`

When you enter only a part of a record without any other command word, such as IS, then only the left side of the IS command is searched.

Format 3: `WHO IS <...>` or `WHAT IS <...>`

This format allows you to search *only* the right side of the IS statement. Note that the words WHO or WHAT are interchangeable.

Renaming Files On The Apple

The best way to rename files on an Apple II family computer using the ProDOS operating system is to employ the FILER program which is provided on your HCJ Volume 4 disk. To use this program, just boot your system with the HCJ disk in the main drive, and select the RUN PRODOS FILER PROGRAM option from the startup menu. Also, if you are not running a program, after having booted with ProDOS, place your HCJ disk in Drive 1, and type **– FILER** at the Applesoft prompt (**>**) to run the program.

Once this program has run, select the FILE COMMANDS option by typing an F, and then select **R** (RENAME FILES). When renaming files on your HCJ Disk, we recommend that you use a backup so you do not inadvertently alter or delete a file from your only copy.

Note that you must enter a complete PATHNAME when selecting both the old and new file names. By using the Set Prefix (P) command from the File Commands menu, you can save some typing in this process. For more information, just type a **?** from the File Commands menu.

Format 4: ALL <...>

This format enables you to search both the left and the right sides of the IS command for the particular piece of information you supply.

Format 5: ALL RELATED TO <...>

This variation on the ALL command (Format 4) also searches both sides of the IS command. When a matching item is found, however, it forces the search to a deeper level. For each entry that matches, the program searches for the item found on the *other* side of the IS command as well, and then finds all entries that match that newly found item. For example, let's say you enter the search parameter **ALL RELATED TO** A BUSINESS ASSOCIATE and your *IS-Base* file contains the entry JOHN PARKER IS A BUSINESS ASSOCIATE. Not only will this particular entry be found, but *all* entries with JOHN PARKER on either side of the IS command will also be found.

2) Change Target Drive

Here you may select single and dual-drive operation. Choosing this option toggles the target drive (the drive in which the file that is merged or built is placed) between your first and second drive. The source disk (the disk containing the original information) always goes into your first drive. (When we refer to first drives, this means drive 1 on the Apple, device 8 on the C-64, drive A on the IBM, and DSK1 on the TI-99/4A. Second drives are drive 2 on the Apple, device 9 on the C-64, drive B on the IBM, and DSK2 on the TI-99/4A.)

Whenever the target drive is set to your second drive, the program is in dual-drive mode—transferring data from the first drive to the second. If you do not own a second drive, do *not* try to use dual-drive mode. If you do own two drives, dual-drive mode saves you from swapping disks in and out of the first drive. The current source and target drive settings are displayed at the bottom of the screen.

3) View Source File

Before you commit yourself to a Build File or Merge File operation, you may want to view your source file. To do this, simply place an *IS-Base* data disk into the source drive and select this option. Notice that only the information matching the current search parameters is listed. This way, you see exactly what information will be used when you Build or Merge a file. If you want to view the entire file, just change your search parameters to an asterisk prior to selecting this option. This option is not limited to viewing source files alone: Any *IS-Base* data disk placed into the source drive will be listed.

To halt a listing, press [ESC] (back-arrow on the C-64, [FCTN] 9 on the TI-99/4A).

4) Build File

Here is where you create new files using an existing file's information. Make sure to set the search parameters prior to selecting this option. You can test the search parameters by viewing the file with the View Source File option.

When you select the Build File option, the computer prompts you to insert your source disk. If you are in dual-drive mode, you are requested to insert your target disk as well. Once you have inserted the proper disks, press the space bar.

As the source disk is searched, any information matching the search parameters is echoed to the screen. When the entire source file has been searched, or the computer's memory is full, the information found is used to build the new file. If you are operating in single-drive mode, you will be required to remove your source disk and put the target disk in its place.

If the target disk already contains an *IS-Base* data file on it, you are given the option to erase the current data file, replacing it with the newly found information, or abort the entire procedure. If all goes well, and you do not abort the procedure, then a new *IS-Base* file is created on the target disk. Single-drive owners may be required to swap disks more than once in order to complete the Build File operation.

The Build File operation comes in handy when creating a new file from a *single* file. If, however, you wish to filter out the same type of information from *several* files, Build File is only the *first* step: After you Build the specialized file, you must Merge the information from the other files, one-by-one, into your newly built file. The next section describes the merging of files.

5) Merge File

Merge File allows you to combine two *IS-Base* files together. By performing several Merge operations, you can combine several files. Again, make sure that the search parameters are properly set before you select this option.

When you select the Merge File option, the computer prompts you to insert your source disk. If you are in dual-drive mode, you are requested to insert you target disk as well. Once you have inserted the proper disks, press the space bar.

As the source disk is searched, any information matching the search parameters is echoed to the screen. When the entire source file has been searched, or the computer's memory is full, the information found is merged with the target file. If you are operating in single-drive mode, you will be required to remove your source disk and put the target disk in its place.

If the target disk does not contain an *IS-Base* file for merging, an error message is displayed and the Merge procedure aborts. If the target disk does contain an *IS-Base* data file, the information found on the source disk is added to the target disk's file. Single-drive users may be required to swap disks more than once in order to complete the Merge File operation.

6) Exit Program

When you are through with your merging and building your new *IS-Base* files, select this option to quit.

Building Files With *IS-Base Construction Set*

To get you started, we've included three new files on your HCJ Volume 4 disk: THOMPSON, BECK, and RODGERS. These *IS-Base* files are quite similar to the MADONNA file included on your HCJ Volume 3 disk. They list information about albums by the Thompson Twins (Here's To Future Days), Jeff Beck (Flash), and Nile Rodgers (B-Movie Matinee). We chose these files for a special reason: In all of them, *and* Madonna's album Like A Virgin, Nile Rodgers played a prominent role in recording and production. (For those of you who may not follow popular music closely, Nile Rodgers is an extremely talented and sought-after producer/musician/songwriter.) And because of this "information overlap" between files, we have suitable material upon which to demonstrate the power of *IS-Base Construction Set*.

We'll start by showing you, step-by-step, how to create new *IS-Base* files from existing ones. Before beginning, you should have a few freshly initialized disks available, so you can freely create new *IS-Base* files without merging or deleting a file you may wish to keep. Remember, because all *IS-Base* files must be called IS-BASE.DAT (ISBASE.DAT on the Apple II, and IS-BASE_DT on the TI-99/4A), only one *accessible IS-Base* file can be on a disk at a time. Of course, you can store several *IS-Base* files on the same disk under *different* names, but to access any particular file, it must have the IS-BASE.DAT file name. For specific information on renaming files on your system, see the appropriate instruction box, *Renaming Files On The* ...

Renaming Files On The C-64

The Commodore 64 operating system allows the renaming of files by accessing what is called the command channel—channel 15. The command to open this channel is the BASIC OPEN command:

```
OPEN15,8,15
```

Once this command has been executed, you use PRINT#15 to access any particular command channel function. Here is the format for a rename command:

```
PRINT#15,"RØ:new name=old name"
```

where new name is the new file name, and old name is the file name you wish to change. You then close the command channel by executing the CLOSE command:

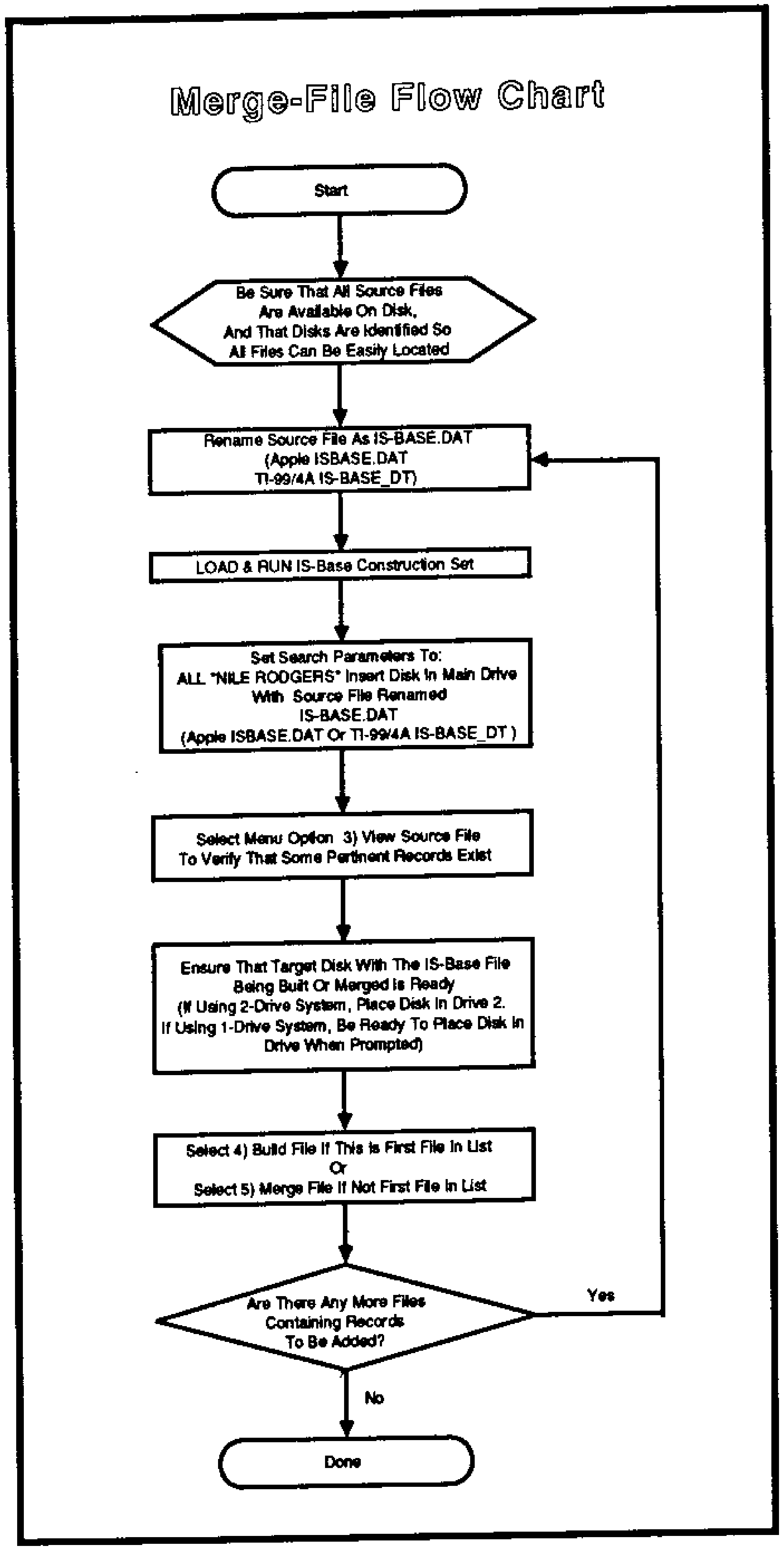
```
CLOSE15
```

For example, to change the file RODGERS to IS-BASE.DAT enter

```
OPEN15,8,1:PRINT#15,"RØ:IS-BASE.DAT=RODGERS":CLOSE 15
```

Be sure that the name you wish to use for the new file name does not already exist on the disk, or the command will not be completed. The only indication you will get that the command was not completed correctly is that the disk drive light will flash.

When renaming files on your HCJ Disk, we recommend that you use a backup so you do not inadvertently alter or delete a file from your only copy. By the way, the *HCJ Duplicator* program, available in HCJ Volume 3, provides an easy way to backup your disks.



Merge-File Flow Chart
Start
Be Sure That All Source Files Are Available On Disk, And That Disks Are Identified So All Files Can Be Easily Located
Rename Source File As IS-BASE.DAT (Apple ISBASE.DAT TI-99/4A IS-BASE_DT)
LOAD & RUN IS-Base Construction Set
Set Search Parameters To: ALL "NILE RODGERS" Insert Disk In Main Drive With Source File Renamed IS-BASE.DAT (Apple ISBASE.DAT Or TI-99/4A IS-BASE_DT)
Select Menu Option 3) View Source File To Verify That Some Pertinent Records Exist
Ensure That Target Disk With The IS-Base File Being Built Or Merged Is Ready (If Using 2-Drive System, Place Disk In Drive 2. If Using 1-Drive System, Be Ready To Place Disk In Drive When Prompted)
Select 4) Build File If This Is First File In List Or Select 5) Merge File If Not First File In List
Are There Any More Files Containing Records To Be Added?
Yes
No
Done

Figure 1.
This Flow Chart is an easy-to-follow guide for creating the Nile Rodgers IS-Base file described in the text. If you follow these instructions, you will extract all the information relating to Nile Rodgers from four IS-Base files: MADONNA (on HCJ Volume 3 disk), ROGERS, THOMPSON, and BECK (on HCJ Volume 4 disk). You will then be able to create a new IS-Base file containing the specialized, related information.

IS-Base Construction Set permits two distinct types of file manipulation: Build File and Merge File. The Build File option allows the creation of a brand new file with parts (or all) of an existing one. For example, let's say you want a file that contains a list of only the people who played guitar on the three Madonna albums. First, you run the program *IS-Base Construction Set*. When the menu appears, place a disk containg the Madonna file (renamed IS-BASE.DAT, of course) in the source disk drive. Next, select menu option 1) Change Search Parameters. The default selection is an asterisk (*), indicating that all records in your *IS-Base* file will be found during a search. Now, replace the asterisk with

```
WHO IS *PLAYING GUITAR*
```

Now the program will find all records that contain the words PLAYING GUITAR on the right side of the IS statement. To verify this, select option 3) View Source File. A list of all the records with the words PLAYING GUITAR will now appear on the screen. To create this new file, select the 4) Build File option. If you have two disk drives and wish to speed up program operation by using both of them, be sure to change the target drive using menu option 2) Change Target Drive, *before* starting to Build the file. You will now be prompted to insert, in turn, your source and target disks at the proper time; the new *IS-Base* file containing all the guitar players on Madonna's three albums is then constructed on the target disk. If you already have an IS-BASE.DAT file on the target disk, you will be asked if you wish to replace the existing file. Only respond Yes if you are sure you want to replace this *IS-Base* file.

Similarly, you can create files of keyboardists, background vocalists, etc. by using similar techniques with these same data files. All you would have to do is change the search parameters to look for the particular type of data you want to place into a file. Remember that *all* the search parameter formats available to you in the *IS-Base* program can be used to build files with *IS-Base Construction Set*.

It is important to realize that *how* you originally entered data determines how easy or difficult building new files will be. When we created the Madonna file for your HCJ Volume 3 disk, we made sure that a similar format was maintained for similar types of data—thus making a search for a particular data type as easy as possible.

Merging Files With *IS-Base Construction Set*

Now that you have had some experience with building new files out of old *IS-Base* files, we will explain the Merge File option. This option is particularly handy if you have several *IS-Base* files, each containing data that you would like merged into a separate file. For an example of this process, we are going to show you how to create a file of all of Nile

Rodgers's contributions to the various albums documented in your four *IS-Base* files. The files containing this information are MADONNA on your HCJ Volume 3 disk, and the files named RODGERS, BECK, and THOMPSON on your HCJ Volume 4 disk.

To follow along with the Merge process that we are about to describe, you will need 3 disks: The first two are for backup copies of Volume 3 and 4 HCJ program disks. We don't recommend using the originals for two reasons: (1) they are write-protected so renaming files is not possible, and (2) even if you could rename files, you should always work with backup disks to avoid destroying data on your original disks. You also will need a new *IS-Base* data disk that you can build and merge your new *IS-Base* file on to. Be sure to label the disks so you can find each of the four files named above.

The basic process goes like this:

1. Rename the data file that you are going to use as the source, IS-BASE.DAT (Apple ISBASE.DAT or TI-99/4A IS-BASE_DT). If you have any questions as to how to rename files on your computer system, see our appropriate instructional box explaining the process in detail.
2. Run *IS-Base Construction Set*.
3. Set the Search Parameters to `ALL *NILE RODGERS*`
4. Select the View Source File Option to ensure that there are pertinent records to be extracted from the Source file. Note: Be sure the correct disk is in the drive when you select this option. The disk must contain the file, and the file must be named correctly.
5. If you have two disk drives available, select option 2 until the target drive is identified on screen as the second drive.

Then be sure the new *IS-Base* data disk in this second drive. If you only have one disk drive, have your data disk ready and place it in the drive when prompted to insert the *target* disk in the drive. You may be prompted to insert the source disk as necessary.

6. If this is the *first* file you are extracting records from, select Build File. If it is the second, third or fourth file, select Merge File.
7. Repeat the process until all data has been extracted.

In our example here, you first rename the MADONNA file to IS-BASE.DAT (ISBASE.DAT on the Apple or IS-BASE_DT on the TI-99/4A) and extract the proper records into your new file. Then you likewise rename RODGERS, extract information, rename BECK, extract information, and finally rename THOMPSON and extract the records from there as well. To help guide you in the process, see the Merge File Flow Chart (Figure 1).

Once you have done the Merge process a few times, you will discover that it is relatively simple, and you should have no trouble using it with any of your own *IS-Base* files.

Renaming Files On The IBM PC

To rename files on an IBM PC, PCjr, Tandy 1000, or other IBM PC compatible computer, use the DOS RENAME command. The format for this command is simply

REN [space] old name **[space]** new name

where old name is the name you wish to change (e.g., RODGERS), and new name is the new file name (e.g., IS-BASE.DAT). Note, that if the name you use for the new name already exists, or the name you use for the old name is not on the disk in drive A:, then you will get a Duplicate file name or File not found error.

When renaming files on your HCJ Disk, we recommend that you use a backup so you do not inadvertently alter or delete a file from your only copy.

Further File Honing

After you go through the process outlined above, you will have extracted every reference to Nile Rodgers in the four files that were supplied on your HCJ Volume 3 and 4 disks. But, what if this is *more* information than you wish to have in one file. Through the judicious use of search parameters, it is easy to zero in on exactly what you wish to isolate. In the example above, you used the parameter `ALL *NILE RODGERS*` to ensure that you extracted all references to Nile Rodgers. If you had desired only references to Nile Rodgers as a song writer or a guitar player, you could have simply used more specific search parameters when searching.

It may prove more efficient, and certainly easier to verify our choices, however, if you first extract information using the general search parameter as we did above, and then Build smaller files out of this one. Let's say that you want a file containing the songs that Nile Rodgers wrote. If you run *IS-Base Construction Set* and place the disk containing our newly created *IS-Base* file in the source drive, you can set the Search Parameter to

```
WHAT IS WRITTEN BY *NILE RODGERS*
```

If you wish to limit your selection to songs written *solely* by Nile Rodgers, you would leave off the wild cards (*), and thus create a file that omits the song PLAN-9 written by Jimmy Bralower *and* Nile Rodgers. In a similar fashion, you can easily create individual files listing albums on which Nile Rodgers plays guitar (WHO IS PLAYING GUITAR*), or produces (WHO IS *PRODUCER*), etc.

If you actually wish to try this operation, you must use a target disk that does not contain an *IS-Base* file with the IS-BASE.DAT file name.

Excluding Data From Files

There is one more procedure for file manipulation which deserves special notice. We have shown you how to create files where you look for a particular type of data, but what if you want everything that's in a file *except* some particular type of data. Here, you use a combination of *IS-Base Construction Set* and your original *IS-Base* program (of Volume 3) to create the new file.

For example, what if you want an *IS-Base* file that includes all of the information about Nile Rodgers, *except* the songs he wrote. You begin by following the process outlined above to create our new Nile Rodgers *IS-Base* file. Then, you run *IS-Base*, and place the disk containing our new *IS-Base* file in the main disk drive and type

```
FORGET WHAT IS WRITTEN BY *NILE RODGERS*
```

The program will find all songs written by Nile Rodgers and will ask if you wish to forget them. By responding Yes, you will create a file that contains all the information about Nile Rodgers excluding any reference to songs he has written.

Be sure to experiment liberally with these sample files. You'll gain good working knowledge of how to get maximum use of *IS-Base* and *IS-Base Construction Set*. Then try the procedures and techniques on your own *IS-Base* files. In no time at all, you're sure to become a Wizard of IS...

Renaming Files On The TI-99/4A

The easiest way to rename files on the TI-99/4A computer is to use the Disk Manager Command Module that came with your Disk Drive system. With the power off, insert the module in the computer, turn on the power to your disk system and Peripheral Expansion Box, then turn on the power to your computer. Next, press any key, and select option **2** Disk Manager from the main menu. Then, choose **1** File Commands, and finally, **2** Rename File. Now, with the disk containing the file you wish to rename in drive 1, select **1** as your master disk. When prompted, enter the name of the file you wish to change (e.g., **RODGERS**), and then the new file name (e.g., **IS-BASE_DT**). To enact the name change, press **[FCTN] 6** (Proc'd).

When renaming files on your HCJ Disk, we recommend that you use a backup so you do not inadvertently alter or delete a file from your only copy.

Never Out Of Sorts

Confused about the quickest way to whip your data into logical order? Here's a collection of the most popular routines to try out—just the sort of information you always wanted to know, but were afraid to ASCII...

One of the most useful functions of the personal computer is its ability to collate, sort, and alphabetize information. For a computer to perform a function, such as sorting in ascending or descending order, it must have a precise, unambiguous procedure to apply to this problem. Such a procedure is called an algorithm.

An algorithm describes a sequence of operations that will, when applied to given information, produce a desired result. In other words, it is simply a recipe or a set of directions. We use algorithms unknowingly every day. Rules for playing a game, road maps, instructions for using your computer, and recipes for cooking are all examples of algorithms.

Directional Algorithms

In order to be useful, an algorithm must be clear, precisely defined, and effective. To illustrate, let's look at good and bad algorithms for locating the drugstore. A clear, precise algorithm would read: "Go west for 3 blocks, then turn right at the traffic light onto Robie Street. Travel on Robie until you come to North Street. Turn left onto North Street, and the drugstore will be immediately on your right, at 111 North Street." An unclear algorithm would sound like this: "Go west for a while. Then turn right for several blocks. The drugstore is just around the corner of North Street." Thus, to be effective, an algorithm must *precisely* specify the sequential procedure to follow in order to accomplish a stated task.

Our purpose here is to consider algorithms (sequences of operations) that will allow the computer to arrange information systematically. Of course, there are all kinds of sorting routines for arranging data (some even in assembly language, which are obviously faster). But for now, we will evaluate 5 routines written in BASIC that will provide you with some practical examples for your own programming efforts. They are the selection sort, bubble sort, Shell sort, heap sort and quick sort. These algorithms range from very simple to quite complex, and from relatively slow to moderately fast. To look at the listings, one might think that the short, simple routines are faster, but in fact, it is the long, complicated listings that are generally the better and faster algorithms.

Selection Sort

The selection sort is a simple, straightforward routine. It consists of a pair of nested **FOR-NEXT** loops—an outer loop, **FOR I=1 to N-1**, and an inner loop, **FOR J=I+1 to N** (see Listing 1). The outer loop takes the first item, and using the inner loop, compares it to every other item in the list, switching each time it finds a lesser value. After completing the inner loop, the outer loop chooses the second item in the list and repeats the sequence until each item has been compared with every other one.

Though it is a simple algorithm, it makes repeated, unnecessary comparisons. This sort always goes through the complete number of passes set in the **FOR-NEXT** loops, regardless of the state of the list. So an already sorted list will still be put through the entire routine as though it were not sorted.

The selection sort is adequate and even preferable for small lists of items because it is so easy to program. But as you can imagine, this sort takes an unbearably long time with lengthy lists (see Tables 1 through 5). But even though it is not efficient for long lists, you will notice that the selection sort serves very well for programs with small data entry.

Bubble Sort

The bubble sort is a very popular routine because it is simple to understand and implement. Unlike the selection sort, it compares only adjacent items, placing them in ascending order. The procedure begins by comparing the first two items in the sequence. If they are out of order, they are exchanged. The procedure continues, comparing the second item with the third, then the third with the fourth, and so on until the sequence is completed. In general, the lower item is moved upward until it is in the correct position. This is called the bubble sort because items which are too low in the sequence will "bubble up" to reach their correct positions.

A flag is used to determine whether any items were exchanged during a pass through the sequence (see Listing 2). At the beginning, **FLAG** is initialized to 0. If an exchange is made during the sequence, then **FLAG** is set to 1. This causes the sequence to be repeated until no exchanges are made, at which point the sorting is completed. The bubble sort, therefore, takes only as many passes as it needs. An already sorted list would require one pass to determine that no exchanges were made.

The bubble and selection sorts are quite simple to understand, but they are slow to use with long lists. The next three sorts to be considered are more complicated algorithms, but they execute at moderately fast speeds.

Shell Sort

The Shell sort, named after its originator D.L. Shell, is similar to the bubble sort but consists of a somewhat more complicated algorithm. Initially a "gap" size is determined at approximately 3/4 of N, where N is the number of items contained in the list. Instead of comparing just the adjacent items, as the bubble sort does, the Shell sort compares items separated by the gap size, exchanging them when necessary. After a complete pass, the size of the gap is cut in half and the process continues. The Shell sort is a considerably faster routine than the bubble or selection sorts because it requires fewer comparisons and exchanges.

Heap Sort

The heap sort is an even more complicated algorithm which involves the use of a binary tree (See Figure 1). The larger items are worked up a "branch," one by one, until they reach the top. When the largest element has reached the top, it is placed in the last element of the array. That branch is then cut off the tree and the algorithm repeats.

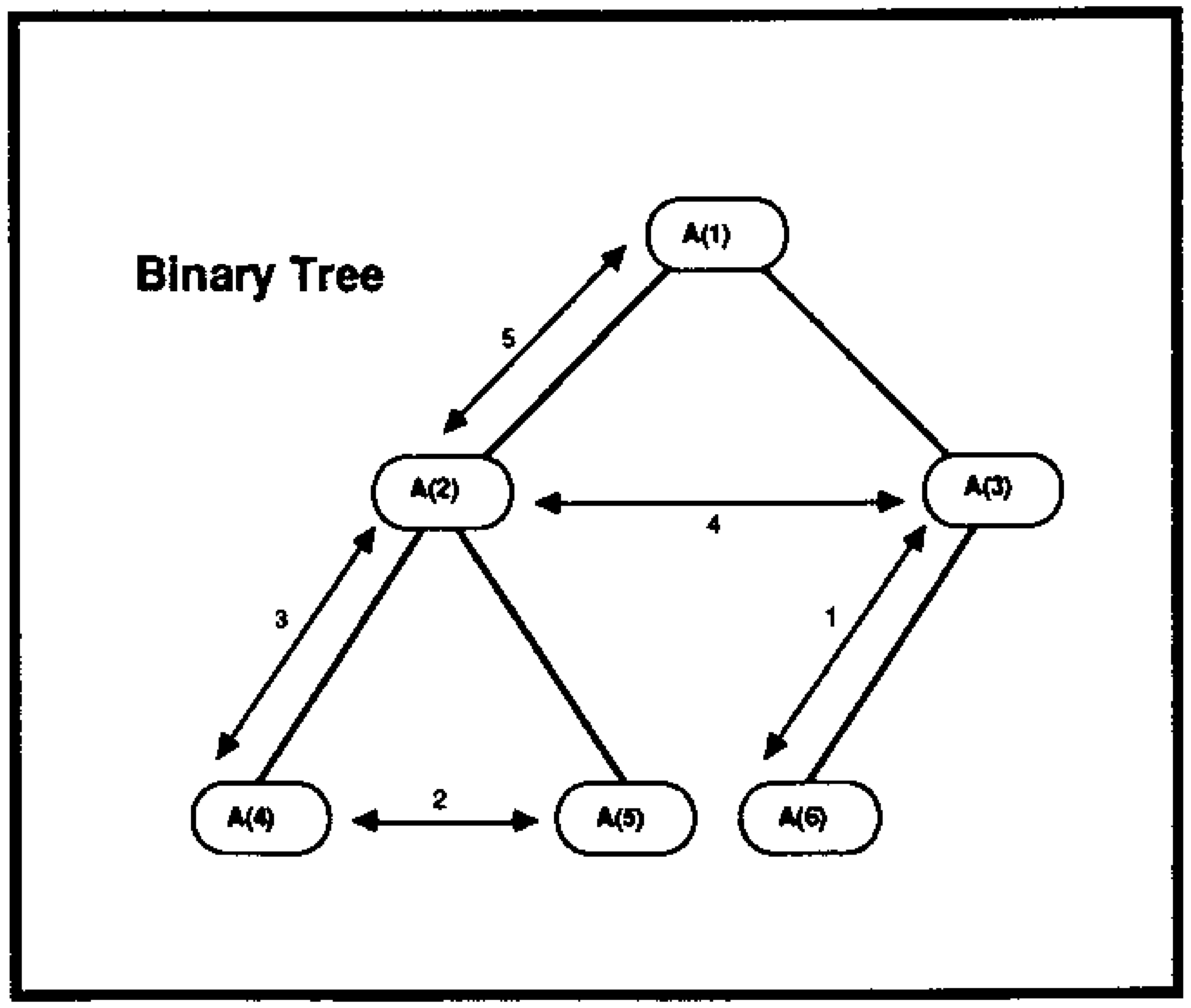


Figure 1.

The following diagram represents the comparisons in one cycle of the heap sort algorithm. The comparisons follow the numbered order, and the interchanges—if necessary—always move the larger value to the array element with the smaller subscript. For example, if A(6) contains 8 and A(3) contains 4, the algorithm puts 8 in A(3) and 4 in A(6). When the largest value reaches A(1), it is exchanged with the value in A(6). Then A(6) is removed from the tree, and the algorithm begins another cycle with A(5).

Quick Sort

The quick sort is generally one of the fastest ways to sort data. It achieves order by first choosing the item on the left end (or bottom) of the list and placing it in its proper place relative to the other items in the list. Then, all the items of lesser value are placed to its left and items of greater value are placed to the right. The list has now been divided into right and left lists. These two lists are repeatedly divided with items being exchanged until the entire array is sorted. Though it is somewhat complicated, the quick sort is a very efficient routine.

Sort Comparisons

Usually one of the major concerns in determining the efficiency of a sorting algorithm is the speed of the sort. Other factors can also be considered, such as the number of comparisons the sort makes and the number of exchanges executed. But for our purposes, we felt a comparison of the sorting speeds would be of most interest and value to the reader. To test them, we generated repeated lists of random numbers, timing the sorting sequence with a stopwatch. The results were averaged and placed in Tables 1 through 4. Note that if lists are nearly in order, the comparitive performance of the algorithms is significantly different. Specifically, the bubble sort is the fastest algorithm if it works on a list that is already in order, because it makes only one pass. All the other sort algorithms make *several* unnecessarry comparisons and swaps on lists that are nearly sorted at the outset.

In our tests on randomly generated lists, the more complex algorithms were, for the most part, significantly faster. It was interesting to note that the simpler routines were nearly as efficient as the others for very small lists of items. But the greater the length of the lists, the more efficient the complex routines became. Sorting 100 items resulted in an 8 to 1 ratio in sort time between the fastest (quick sort) and the slowest (bubble sort). It's important to note that although these programs sort *numbers*, the same program logic can also be used to sort *alphabetically*. All you need to do is to change the appropriate variables to string variables.

Programs On Disk

We have provided these five sort routines on your HCJ Volume 4 disk under the file names SELECT, BUBBLE, SHELL, HEAP, and QUICK. Each of these programs begins by randomly selecting 100 items and printing them on the screen (program lines 100-190). Next, the appropriate sort routine is called and the sorted numbers are output to the screen (lines 200-280). These are the actual sort routines we used to compile the benchmarks shown in Tables 1-5.

In each of these programs, the actual sort routines are stored as subroutines starting at line number 1000, and these subroutines are shown in Listings 1-5. If you wish to use any of these routines in your own programs, you only need to use the subroutine starting at line

1000, and do a GOSUB to the routine with the list of items to be sorted in the A( ) array, and the number of items to be sorted in the variable N. Once completed, the routine will return with the items sorted— the smallest value in element A(1) and the largest value in element A(N).

One final note: These routines were written so they could run using any of the BASIC languages of the computers we cover. You may find ways of streamlining the code to best suit your computer's BASIC. Take care, however, that in making modifications you do not alter the function of an underlying algorithm.

Listing 1
Selection Sort

```
1000 REM **SELECTION SORT**
1010 FOR I=1 TO N-1
1020 FOR J=I+1 TO N
1030 IF A(I)<=A(J) THEN 1070
1040 CHANGE=A(I)
1050 A(I)=A(J)
1060 A(J)=CHANGE
1070 NEXT J
1080 NEXT I
1090 RETURN
```

Listing 2
Bubble Sort

```
1000 REM **BUBBLE SORT**
1010 FLAG=0
1020 FOR I=1 TO N-1
1030 IF A(I)<=A(I+1) THEN 1080
1040 CHANGE=A(I)
1050 A(I)=A(I+1)
1060 A(I+1)=CHANGE
1070 FLAG=1
1080 NEXT I
1090 IF FLAG=1 THEN 1010
1100 RETURN
```

Listing 3
Shell Sort

```
1000 REM **SHELL SORT**
1010 GAP=N*1.5
1020 GAP=INT(GAP/2)
1030 IF GAP=0 THEN 1150
1040 FOR I=1 TO N-GAP
1050 J=I
1060 K=J+GAP
1070 IF A(J)<=A(K) THEN 1130
1080 CHANGE=A(J)
1090 A(J)=A(K)
1100 A(K)=CHANGE
1110 J=J-GAP
```

Listing 4
Heap Sort

```
1000 REM **HEAP SORT**
1010 K=N
1020 L=INT(N/2)+1
1030 IF L=1 THEN 1070
1040 L=L-1
1050 S=A(L)
1060 GOTO 1130
1070 S=A(K)
1080 A(K)=A(1)
1090 K=K-1
1100 IF K>=1 THEN 1130
1110 A(I)=S
1120 GOTO 1270
1130 J=L
1140 I=J
1150 J=J+J
1160 IF J<=K THEN 1190
1170 A(I)=S
1180 GOTO 1030
1190 IF J>=K THEN 1220
1200 IF A(J)>=A(J+1) THEN 1220
1210 J=J+1
1220 IF S<A(J) THEN 1250
1230 A(I)=S
1240 GOTO 1030
1250 A(I)=A(J)
1260 GOTO 1140
1270 RETURN
```

Listing 5
Quick Sort

```
1000 REM **QUICK SORT**
1010 P=1
1020 L(P)=1
1030 R(P)=N
1040 IF P<=0 THEN 1410
1050 LB=L(P)
1060 RB=R(P)
1070 P=P-1
1080 IF RB<=LB THEN 1040
1090 I=LB
1100 J=RB
1110 T=A(I)
1120 IF J<1 THEN 1160
1130 IF T>=A(J) THEN 1160
1140 J=J-1
1150 GOTO 1120
1160 IF J>I THEN 1190
1170 A(I)=T
1180 GOTO 1310
1190 A(I)=A(J)
1200 I=I+1
1210 IF I>N THEN 1250
1220 IF A(I)>=T THEN 1250
1230 I=I+1
1240 GOTO 1210
1250 IF J<=I THEN 1290
1260 A(J)=A(I)
1270 J=J-1
1280 GOTO 1130
1290 A(J)=T
1300 I=J
1310 P=P+1
1320 IF I-LB>=RB-I THEN 1370
1330 L(P)=I+1
1340 R(P)=RB
1350 RB=I-1
1360 GOTO 1080
1370 L(P)=LB
1380 R(P)=I-1
1390 LB=I+1
1400 GOTO 1080
1410 RETURN
```

Note: See the following page for Tables 1-5 which compare the execution times of the various sort routines.

Tables 1-5.

These tables show the relative performance in seconds of the routines shown in Listings 1-5. Use these as a general guide when trying to determine which sort would be most suitable for your own program. Also, it is important to realize that the lists used for these benchmarks were all generated randomly. If the lists were already sorted the comparative performance will undoubtedly differ. For example, if a test list had just one out-of-place item, some of the routines that appear inefficient here may actually complete this simple sorting task more quickly than one of the "more efficient" sorts.

Table 1:

Apple II

List Size	Selection	Bubble	Shell	Heap	Quick
10	000.8	001.5	000.7	001.3	001.3
25	004.7	008.5	002.7	004.2	003.2
50	017.5	034.3	006.9	010.5	007.1
100	066.2	136.0	017.5	024.8	015.5
500	forget it	forget it	125.1	170.0	101.8

Table 2:

C-64

List Size	Selection	Bubble	Shell	Heap	Quick
10	000.8	001.4	000.6	001.3	001.3
25	004.8	009.0	002.7	004.1	002.9
50	018.1	030.2	007.0	010.1	006.9
100	071.0	140.5	017.8	023.1	015.3
500	forget it	forget it	125.2	156.8	100.7

Table 3:

IBM PC

List Size	Selection	Bubble	Shell	Heap	Quick
10	000.7	001.1	000.7	001.0	000.8
25	003.8	006.5	002.2	002.7	002.3
50	014.0	025.1	005.5	006.9	004.9
100	054.1	099.4	013.2	016.4	010.9
500	forget it	forget it	092.3	109.9	076.8

Table 4:

IBM PCjr

List Size	Selection	Bubble	Shell	Heap	Quick
10	000.7	001.4	000.7	001.1	000.9
25	004.5	008.2	002.8	003.4	002.8
50	017.2	029.4	006.9	008.5	006.1
100	064.8	135.5	017.2	019.8	013.2
500	forget it	forget it	114.7	134.5	085.3

Table 5:

TI-99 4A

List Size	Selection	Bubble	Shell	Heap	Quick
10	001.4	001.9	001.2	001.8	001.7
25	006.5	010.4	004.0	005.2	009.4
50	024.3	045.9	010.2	012.5	023.0
100	089.9	186.0	024.4	028.9	023.0
500	forget it	forget it	169.7	194.0	142.7

Did You Know That...?

Did you know that you can use Extended BASIC's **ACCEPT** command to input up to 255 characters? Most people don't know it, but it is possible to input more than just one screen line with the TI's versatile **ACCEPT** command. As long as you do not use the **AT** or **SIZE** options, the **ACCEPT** command inputs up to 255 characters. Try it!

Did you know that arrays do not always have to be **DIM**ensioned? If your program uses an array with 10 or less elements, it does not have to be initialized with the **DIM** command—even if it is an array of more than one dimension (i.e., **A(10,10)**).

Did you know that the TI has a built in word-wrap feature? Word wrap is when a word that does not fit at the end of a screen line is printed on the next line to avoid breaking the word into two parts. Both TI BASIC's **PRINT** command and TI Extended BASIC's **DISPLAY** command have this ability. Before printing, the length of all variables and string literals (anything placed inside quotes) is checked. If the item being printed does not fit entirely on the specified line, it will be printed on the following line instead. To experience this feature first hand, type in and run the following code:

```
10 INPUT "ENTER ANYTHING: ";S$
20 PRINT "THIS IS WHAT YOU ENTERED: ";S$
30 IF S$<>"" THEN 10
```

Unless you enter something that is two characters or less in length, **S$** is automatically printed on the line following the string literal **"THIS IS WHAT YOU ENTERED: "**. If you would like to avoid this wrapping feature, use ampersands (&) in place of semicolons (;) when separating the items that you are printing. For instance, try replacing the semicolon in line 20 of the previous example and re-run the program. See the difference? This feature works the same with the **DISPLAY** command as it does with the **PRINT** command.

Did you know that you can use TI BASIC's **INPUT** command without producing the question-mark prompt? In fact, the **INPUT** command allows you to supply your own prompt, be it a null string, a single character, or a an entire sentence. Here are three such examples:

```
10 INPUT "":S$
10 INPUT ">":S$
10 INPUT "ENTER YOUR AGE: ":S$
```

Of course, you can always let the **INPUT** command provide a question mark for you using the standard format shown below.

```
10 PRINT "HOW OLD ARE YOU";
20 INPUT S$
```

Did you know that you can quickly and easily clear specific screen lines using Extended BASIC's **DISPLAY AT** command? By using the **DISPLAY AT** command alone (without supplying the command with anything to print), the computer simply clears the desired screen line. For example, the following code clears the TI's third screen line:

```
10 DISPLAY AT(3,1)
```

With this knowledge, it is easy to see how one might clear several lines at once. This code clears screen lines 10 through 15:

```
10 DISPLAY AT(10,1): : : : : :
```

Note that each colon is separated by a space.

Did you know that you can learn more about the TI-99/4A with each Volume of Home Computing Journal? It's a fact ... so don't forget to order any back volumes you may have missed.

Pull-Down Menus

Order up some pull-down menus for your TI computer —the perfect choice for byte-conscious gourmets…

More and more computer software employs pull-down menus to simplify the interface between user and computer. Computers such as the Apple Macintosh, Commodore Amiga, and Atari ST have made pull-down menus an integral part of their operating systems. Although pull-down menus are not a standard feature on the TI-99/4A computer, they can be added with relative ease—that is, when you use the accompanying program—*TI Menus*.

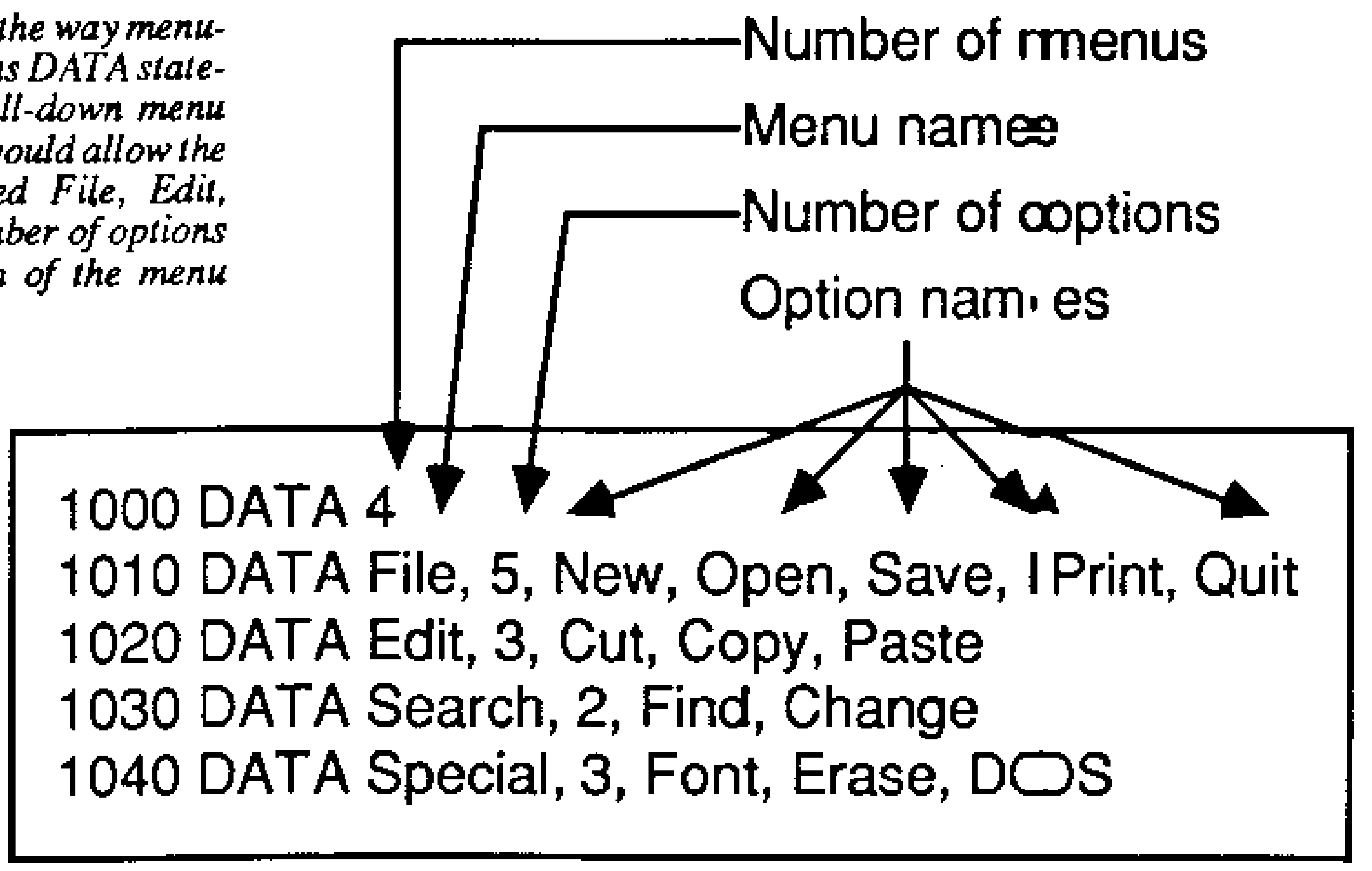
TI Menus is a package of subroutines that add pull-down menus to any Extended BASIC program. With these subroutines, you can define menus and their options, display the menu bar (a list of available menus), and most importantly, allow the user to pull down (display) a menu and select an option. *TI Menus* is supplied on your HCJ Volume 4 disk under the file name of MENU_MERGE. This is a Merge file that you can merge into memory with the following command: **MERGE DSK1.MENU_MERGE**

Oh Walter, Could I Have A Menu Please?

To define your menus, you must enter certain information into **DATA** statements. Look at lines 30290 to 30340 of *TI Menus* to see an example. Here we have entered some sample data. Refer to these **DATA** statements while reading the explanation of their format.

The first piece of data specifies the number of menus that will appear on the menu bar. The data that follows provides the actual menu information. This data consists of the menu name, the number of options in the menu, and then the option names. This information repeats for each menu. Figure 1 summarizes the menu data format.

Figure 1.
Here is an example demonstrating the way menu-setup information could be stored as DATA statements in a program using our pull-down menu routines. These DATA statements would allow the routine to create 4 menus called File, Edit, Search, and Special—with the number of options and option names following each of the menu titles.



```
1000 DATA 4
1010 DATA File, 5, New, Open, Save, Print, Quit
1020 DATA Edit, 3, Cut, Copy, Paste
1030 DATA Search, 2, Find, Change
1040 DATA Special, 3, Font, Erase, DOS
```

With the **DATA** statements all set up, you're ready to install pull-down menus in a program. There are three main subroutines in *TI Menus*. These subroutines should be called in the order in which they are listed here. The first subroutine (**GOSUB 30080**) initializes the pull-down menus by loading the machine-code file GETPUT_O, reading the menu data, and setting up some key variables. You must make sure that the computer's **DATA** pointer is **RESTORE**d to your menu data when you call this subroutine.

The second subroutine (**GOSUB 30350**) simply displays the menu bar at the top of the screen.

The third subroutine (**GOSUB 30540**) operates the pull-down menus. When called, the user is allowed to pull down any of the menus and select an option. There are two variables that are passed back from this last routine—the variables **MENU** and **OPT**. **MENU** is set equal to the number of the menu from which a selection was made. This number corresponds to the position of the menu on the menu bar. **OPT** is set equal to the number of the option that was selected. This number corresponds to the position of the option within the menu. If the user did not select an option, **OPT** will equal zero. See Figure 2 for a list of these **GOSUB**s and their function.

TI Menus uses the commands **GETSTR** and **PUTSTR** (published in the TI Tech Note for HCJ Volume 3) to create pull-down menus. *TI Menus* must load the file GETPUT_O in order to install these two commands. If the file GETPUT_O is not in DSK1 when the program is run, you will get an error message. (GETPUT_O is included on your HCJ Volume 4 disk.) So, GETPUT_O should be placed on all disks containing programs that makes use of *TI Menus*.

I'm Ready To Order Now

Making selections from the menu is easy. When the menus are made operative (via a **GOSUB 30540**), the first menu on the menu bar is automatically pulled down and made active. To select an option, use the **[FCTN] E** and **[FCTN] X** keys. To move from one menu to the next, use the **[FCTN] S** and **[FCTN] D** keys. As you move from menu to menu, the old menu will close as the new one is pulled down. Note that any screen information that is covered up by a menu is restored when the menu is closed. Pressing **[ENTER]** makes your menu selection final. If you decide not to choose any of the options, press **[FCTN] 9**.

Considering The Menu's Right-Hand Column

As with most things in life, there is a price you have to pay when using *TI Menus*. Fortunately, the price is small—only about 3K. *TI Menu's* code and variables take up approximately 3K of the BASIC workspace (depending on the number and size of menus defined). This is not a major chunk, but something to consider when using *TI Menus* within a large program.

The menu bar takes up the top two lines on the computer screen, so avoid placing any text there. Also, please take into account the width of the screen when defining your pull-down menus. If you define too many menus, or use names that are too long to fit on the screen, *TI Menus* will not work properly.

TI Menus uses 12 variables (see Key Variables). Avoid using variables of the same name in your host program. One more thing to consider: *TI Menus* is contained in line numbers 30000 to 30670. If your program uses these line numbers, you must renumber *TI Menus*.

Today's Special

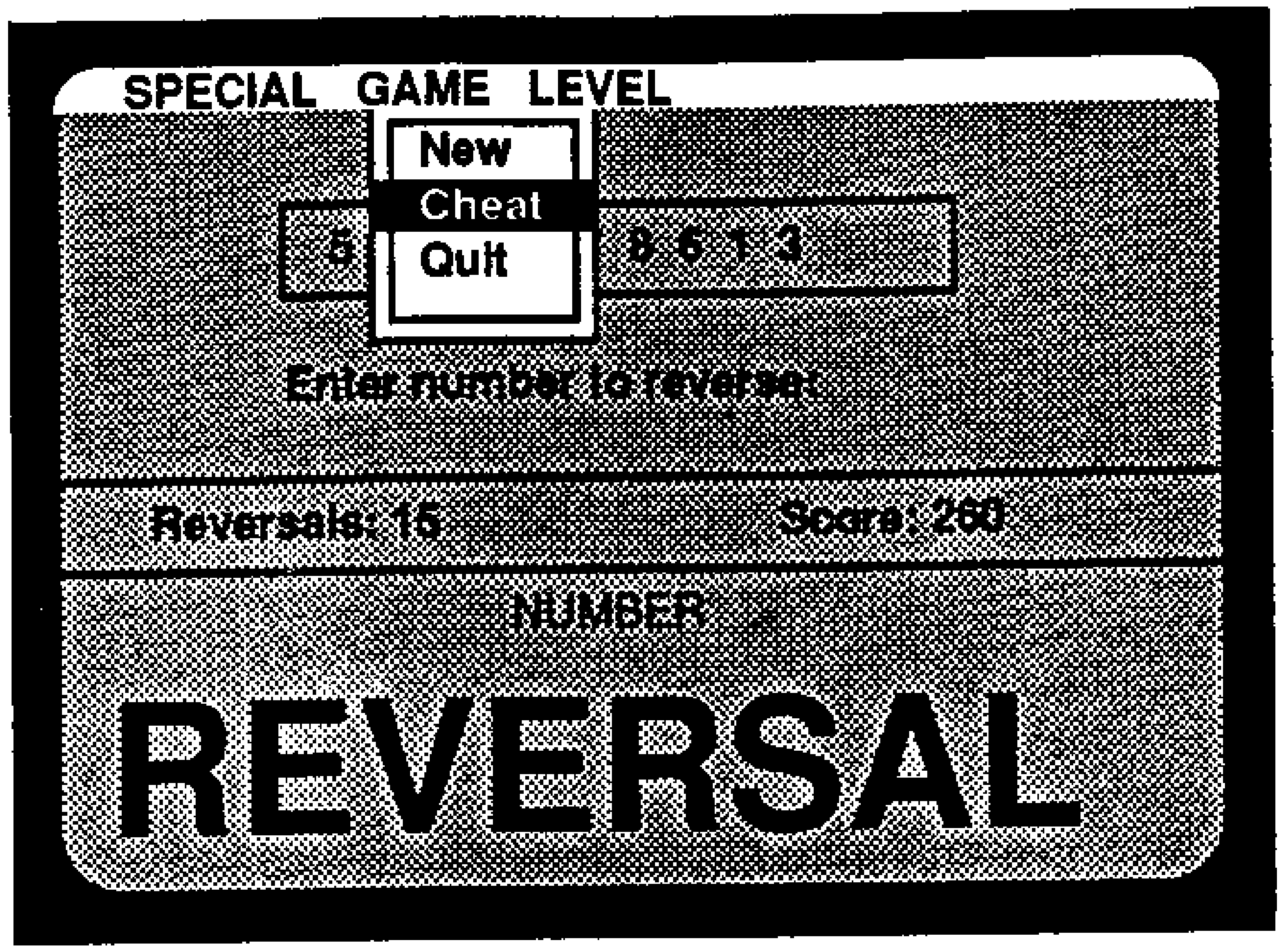
On your HCJ Volume 4 disk is a program called *Number Reversal*. It is saved under the file name of REVERSAL; the program's Control Capsule appears on this page. This program takes advantage of the pull-down menus provided by *TI Menus*. It is a good example of how to use *TI Menus* in a program. Line 230 initializes the menus, line 240 displays the menu bar, and line 370 activates the menus whenever [CTRL] M is pressed.

Key Variables

TI Menus

Variable	Function
NUMMENU	Number of menus
NUMOPT( )	Number of options for each menu
MENUBAR$	String of menu names
MENU$	String of menu option names
MENUX( )	Horizontal position of each menu
OPTLEN( )	Length of option names for each menu
SCREEN$	String to hold screen data
MENU	Menu from which selection was made
OPT	Option that was selected
LINE$	Multi-purpose string variable
MM	Current menu showing
MO	Current option highlighted

Number Reversal is a game of logic in which you are presented with a series of digits. You goal is to rearrange the digits into numeric order. This may seem like a menial undertaking at first, but wait until you play the game... To reorder the numbers you must *reverse* the order of selected digits. To choose the number of digits to reverse, select one of the digits. Once selected, all digits to the left of, and including the selected digit are reversed. Sound confusing? Don't worry—it's supposed to be! Just run the program and experiment. If you're persistent, you'll get it.



5 8 6 1 3

Enter number to reverse:

Reversals: 15 Score: 260

Figure 2.

Subroutine	Function
GOSUB 30080	Initialize menu routines
GOSUB 30350	Display menu bar
GOSUB 30540	Operate pull-down menus

Control Capsule

Number Reversal

Key	Function
0 through 9	Select digit to reverse
CTRL M	Use pull-down menus

Using Pull-Down Menus

FCTN S	Move to the menu on the left
FCTN D	Move to the menu on the right
FCTN E	Select previous option
FCTN X	Select next option
ENTER	Make menu selection final
FCTN 9	Exit pull-down menus without making selection

IBM PC, PCjr, and Tandy 1000

Procedures For Using The IBM PC, PCjr, Or Tandy 1000

To make use of the HCJ Director menu program on your HCJ disk you need to backup your disk. Use the following procedures to produce an autoboot backup of your HCJ disk:

If you have a dual-drive system you may start with step 1, otherwise read this paragraph first:

For those of you with a single disk drive, you may still use the commands as listed below, though you will need to pay very close attention to the prompts on the screen instructing you to swap disks from time to time. The computer will tell you to place the appropriate disk in drive B:. What it means, however, is to remove the disk from drive A: and insert the disk which would have gone in drive B:. Using a single drive may mean having to swap disks quite a few times. For those who are patient though, the rewards are worth the added work. If you have further questions consult your DOS manual on the FORMAT and COPY procedures for a single-drive system.

1. Place your DOS master disk (hereafter refered to as the DOS disk) in drive A: and turn on the power to your system.
2. Enter the command **FORMAT B: /S /V**
3. The computer will ask you to place a blank disk into drive B: to be formatted. Ensure that the blank disk is in the drive and then press [**Enter**]. After formatting, you will be asked for a volume name. Enter **HCJOURNALn** where **n** is the Volume number. Then, you will be asked if you want to format another. Respond No to this prompt which returns you back to DOS.
4. If you wish to use a color monitor enter the command **COPY A:MODE.COM B:**
5. Enter the command **COPY A:BASIC*.* B:BASIC.***
If you have an IBM compatible whose BASIC does not start with the word BASIC, then make adjustments in the command above for your version. In any case, the file on your new boot disk should always be named BASIC even if it was originally named BASICA.
6. After BASIC is copied to the new disk in drive B:, remove the DOS disk from drive A: and place the HCJ disk in drive A:
7. Enter the command **COPY A:\*.* B:**
8. After the last file has been copied, remove both disks from the system. Label the new disk as **HCJ ON DISK BACKUP Volume n**, where **n** is the Volume number, and place the original disk in a safe place.
9. The new disk you have created can now be used to boot your system (start from a power off condition) and will automatically bring up a menu of programs from which you may select.

You may also use the HCJ disk without backing it up if you:
1. Start from DOS 2.1 or later.
2. If you wish to run a program with a BAT, COM, or EXE extension, simply type the file name from the DOS A> prompt.
3. If you wish to run a BASIC program, you must first enter the appropriate version of BASIC, then **LOAD** and **RUN** the program.
Note: If you have an IBM PC and the program requires a color monitor, you must enable the monitor using the appropriate DOS **MODE** command before running the program.

Program Name	**File Name**	**Language**
HCJ Director	HCJDIR.COM*	Turbo Pascal
	AUTOEXEC.BAT	-batch file-
CodeWorks	CODEWORK.COM*	Turbo Pascal
IS-Base Construction Set	ISCONST.COM	C-64 BASIC
	BECK	-data file-
	RODGERS	-data file-
	THOMPSON	-data file-
Grid Grappler	GRID.BAS**	BASICA
Never Out Of Sorts	SELECT.BAS**	BASICA
	BUBBLE.BAS**	BASICA
	SHELL.BAS**	BASICA
	HEAP.BAS**	BASICA
	QUICK.BAS**	BASICA
Pascal Sorts	SELECT.PAS	Turbo Pascal Source
	SELECT.COM	Turbo Pascal
	BUBBLE.PAS	Turbo Pascal Source
	BUBBLE.COM	Turbo Pascal
	SHELL.PAS	Turbo Pascal Source
	SHELL.COM	Turbo Pascal
	HEAP.PAS	Turbo Pascal Source
	HEAP.COM	Turbo Pascal
	QUICK.PAS	Turbo Pascal Source
	QUICK.COM	Turbo Pascal
Basic Batch Processor	BATCH.BAS**	BASICA
	KEYS.TXT	-data file-
	SCREEN.TXT	-data file-

*Program requires: DOS 2.1 or later.

**Program requires: DOS 2.1 & either Cartridge BASIC on PCjr or BASICA on PC, or GW BASIC on Tandy 1000.

TI-99/4A

Procedures For Loading The TI-99/4A With Extended BASIC

1. Ensure the Peripheral Box is properly connected to the console. Turn on the Peripheral Box.
2. Place the Extended Basic module securely in the machine.
3. Turn on the TI-99/4A computer.
4. Insert the HCJ disk into drive 1.
5. Strike any key to bring up the first menu, then select Extended BASIC, and The HCJ Director program will automatically **RUN**.
6. Select the number of the program you wish to run, then press [**ENTER**] and the program will load and **RUN** automatically.

Procedures For Loading The TI-99/4A With TI BASIC

1. Ensure the Peripheral Box is properly connected to the console. Turn on the Peripheral Box.
2. Turn on your computer and insert the HCJ disk in drive 1.
3. Strike any key to bring up the first menu, then select BASIC.
4. To load the BASIC program you wish to use, type **OLD DSK1.file name** where **file name** is the file name of the program. For example, if you wish to use *Perfect Puppy* type **OLD DSK1.PERFECT** and press [**ENTER**]. Now, type **RUN** and press [**ENTER**].

Program Name	**File Name**	**Language**
HCJ Director	LOAD	Extended BASIC
Codeworks	CODEWORK	Extended BASIC
IS-Base Construction Set	ISCONST*	Extended BASIC
	BECK	-data file-
	RODGERS	-data file-
	THOMPSON	-data file-
Never Out Of Sorts	SELECT	Extended BASIC
	BUBBLE	Extended BASIC
	SHELL	Extended BASIC
	HEAP	Extended BASIC
	QUICK	Extended BASIC
Grid Grappler	GRID	Extended BASIC
Pull-Down Menus	MENU_MERGE*	Extended BASIC
	GETPUT_O*	TMS 9900 object file
	REVERSAL*	Extended BASIC

*Requires 32K Memory expansion.

HCJournal™

Don't Miss A Single Volume
—Subscribe/Renew Today...
...And Don't Forget
To Tell Your Friends!

Home Computing Journal (HCJ) is a quarterly multi-media software subscription service containing ready-to-run productivity, education, entertainment, and utility programs on a floppy disk. The accompanying workbook contains the required support documentation plus additional technical notes and programming aids.

Artificial intelligence, database management, high-powered programming aids, realistic simulations, and specialized software for personal investing, task-specific report writing, computer-assisted design, desktop publishing, personal communications, plus entertaining math and logic excursions are just some of the projects already on our planning board.

YES, Please accept my order for

Home Computing Journal:

☐ New purchaser ☐ Renewal purchaser

☐ HCJ Anthology: **$75** postpaid U.S
(Volumes 1-4) U.S. **$89** in Canada

☐ Single-Volume Price: **$25** postpaid U.S.
U.S. **$30** in Canada

Each quarterly Volume of HCJ includes the printed Journal plus the companion disk for your computer choice indicated below.

DISK VERSION					
APPLE	ATARI*	C-64	IBM PC	IBM PCjr	TI

* Only Atari back-Volumes 1-3 available

Please indicate Volume Nos. Ordered ☐☐☐
Each Volume is numbered sequencially—i.e., Volume 1, Volume 2, Volume 3...

TOTAL ORDER: ____________

☐ Check or Money Order Enclosed
MUST BE IN U.S. FUNDS DRAWN ON A U.S. BANK

Charge my: ☐ VISA ☐ MasterCard

Account No. ☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐

DATE EXPIRES __________ SIGNATURE ____________________

PLEASE PRINT ALL INFORMATION BELOW

NAME ______________________________

ADDRESS ______________________________

CITY ____________________ STATE ___ ZIP ________

TEL. NO. ______________________________

Prices Subject To Change Without Notice.

Mail with check, money order, or credit card information to:

Home Computing Journal
P.O. Box 70248 • Eugene, OR 97401
MC/VISA orders may also be placed by telephone:
Tel. (503) 342-4013
West Coast Time (Normal Business Hours)

If you prefer not to cut this copy of your Journal, you may photocopy this form to send in with your order.

HCJournal™

Don't Miss A Single Volume
—Subscribe/Renew Today...
...And Don't Forget
To Tell Your Friends!

Home Computing Journal (HCJ) is a quarterly multi-media software subscription service containing ready-to-run productivity, education, entertainment, and utility programs on a floppy disk. The accompanying workbook contains the required support documentation plus additional technical notes and programming aids.

Artificial intelligence, database management, high-powered programming aids, realistic simulations, and specialized software for personal investing, task-specific report writing, computer-assisted design, desktop publishing, personal communications, plus entertaining math and logic excursions are just some of the projects already on our planning board.

YES, Please accept my order for

Home Computing Journal:

☐ New purchaser ☐ Renewal purchaser

☐ HCJ Anthology: **$75** postpaid U.S
(Volumes 1-4) U.S. **$89** in Canada

☐ Single-Volume Price: **$25** postpaid U.S.
U.S. **$30** in Canada

Each quarterly Volume of HCJ includes the printed Journal plus the companion disk for your computer choice indicated below.

DISK VERSION					
APPLE	ATARI*	C-64	IBM PC	IBM PCjr	TI

* Only Atari back-Volumes 1-3 available

Please indicate Volume Nos. Ordered ☐☐☐
Each Volume is numbered sequencially—i.e., Volume 1, Volume 2, Volume 3...

TOTAL ORDER: ____________

☐ Check or Money Order Enclosed
MUST BE IN U.S. FUNDS DRAWN ON A U.S. BANK

Charge my: ☐ VISA ☐ MasterCard

Account No. ☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐

DATE EXPIRES __________ SIGNATURE ____________________

PLEASE PRINT ALL INFORMATION BELOW

NAME ______________________________

ADDRESS ______________________________

CITY ____________________ STATE ___ ZIP ________

TEL. NO. ______________________________

Prices Subject To Change Without Notice.

Mail with check, money order, or credit card information to:

Home Computing Journal
P.O. Box 70248 • Eugene, OR 97401
MC/VISA orders may also be placed by telephone:
Tel. (503) 342-4013
West Coast Time (Normal Business Hours)

If you prefer not to cut this copy of your Journal, you may photocopy this form to send in with your order.