TURBO GAMEWORKS

Source Code, Secrets, & Strategies Included

CHESS, BRIDGE & GO-MOKU

TURBO GAMEWORKS

Owner's Handbook

BORLAND INTERNATIONAL Inc.
4585 Scotts Valley Drive
Scotts Valley, CA 96066
U.S.A.

TABLE OF CONTENTS

APPENDICES

LIST OF FIGURES

INTRODUCTION

Welcome to the Turbo GameWorks package for the IBM PC and compatible computers. The software and this manual can enhance your understanding of how to design and program strategic games. The games themselves are also a lot of fun. You can brush up on your Turbo Pascal technique, learn game programming by example, or just sit back and enjoy playing.

To get full benefit from the Turbo GameWorks package, you should be familiar with the Pascal programming language (another programming language can be of help, too, but Pascal is best). *Turbo Tutor* (available from Borland) can get you up to speed in Pascal in a hurry. If your interest is only in playing the games, however, you don't need to know anything about programming.

Computer game programs and what goes on inside them have been, for the most part, the well-kept secret of hobbyists and professional game designers. While we don't divulge any trade secrets here, we do reveal some tricks of the trade for the aspiring game designer.

Turbo GameWorks contains well-commented Pascal source code for three games: go-moku, chess and bridge. The manual walks you through the programs that play each of these games, and provides tips and comments for building computer games in general.

The algorithms and programming techniques in the Turbo GameWorks package have many potential applications in business and decision-making software. We hope that by revealing some of these heretofore secret techniques, Borland will spur new developments in productivity software.

Shake Hands with Your Computer Opponent

What would a games development package be if you couldn't play? The three games in Turbo GameWorks are all very playable. Each game is introduced with a brief set of rules written by an expert on each topic, followed by the playing instructions.

The games are provided in fully compiled .COM files, so that you won't have to wait even the few seconds that Turbo Pascal would take to compile each program. Turbo GameWorks also provides full source code ready for you to examine, change, and analyze.

What's Inside Turbo GameWorks?

Turbo GameWorks takes three well-known games apart for you, to show you what makes them work. We'll cover:

- *Evaluating moves*—How does a computer game know where to move next? How does it assess your moves? The evaluator is the heart of every computer game.
- *Data structures*—How does a game designer begin to think about representing the playing field, the pieces and the moves of a strategic game?
- *Search strategy*—The three fundamental search techniques that help you speed the execution of your games, and the pros and cons of each.
- *User interface*—A game is no good if the computer has all the fun! We'll look at Pascal procedures that trap user input, and the aesthetics of screen layout and design.

Structure of this Manual

The body of the manual is divided into two main parts:

- Section I—*Playing Turbo Games* provides specific instructions for using each game program.
- Section II—*Into the Source Code* provides a guided tour of each game program, beginning with a description of playing strategy and how that translates into program design and the application of the playing algorithms. Then, the Pascal procedures that affect the play of each game are described.
- Appendix A provides a short history of computer chess.
- Appendix B contains the official rules of chess.
- Appendix C covers the rules of bridge.
- Appendix D gives a list of suggested books about chess, bridge and Go-Moku.
- Appendix E is a glossary of terms you need to understand the games and this manual.

Throughout the manual, design hints, programmer's tips and anecdotes that might help your understanding or expand your game programming horizons are boxed or set apart in the margins separate from the main text.

Typography

The body of this manual is printed in normal typeface. Special characters are used for the following special purposes:

`Alternate` `An alternate typeface is used in program examples and procedure and function declarations0`

Italics *Italics are used to emphasize certain concepts and terminology, such as predefined standard identifiers, parameters and other syntax elements.*

Boldface **Boldface type is used to mark reserved words in the text as well as in programming examples.**

Please refer to the *Turbo Pascal Reference Manual* for a complete description of the syntax, special characters, and overall appearance of the Turbo Pascal language.

The Distribution Diskettes

The Turbo GameWorks package is contained on two diskettes. Turbo Go-Moku and Turbo Bridge are contained on one diskette, and Turbo Chess on the other. Each game consists of .COM, .PAS, .CH (for Turbo Chess), .BR (for Turbo Bridge) and help files. Run the README.COM program for a complete list of these files, and for any last minute information not contained in this manual.

To fully benefit from Borland's update and support policy, please complete and mail the license agreement at the front of this manual.

Acknowledgments

- Turbo Pascal, Turbo Tutor, and Turbo Graphix Toolbox are trademarks of Borland International
- Pente is a trademark of Parker Brothers
- Zork is a trademark of Infocom

Section I

PLAYING TURBO GAMES

Chapter 1
PLAYING GO-MOKU

Go-Moku is a simpler version of the ancient Japanese game of Go, and it also closely resembles the contemporary game of Pente. The two players (in this case, the human player and the computer) take turns placing 0's and X's on the intersections of a 19 × 19 line grid. The object of the game is to line up five pieces in a row. Watch out—this game is deceptively simple. While Go-Moku isn't nearly as complex as Go, it's also not as easy as it first appears. For all its simplicity, Go-Moku can be a real challenge.

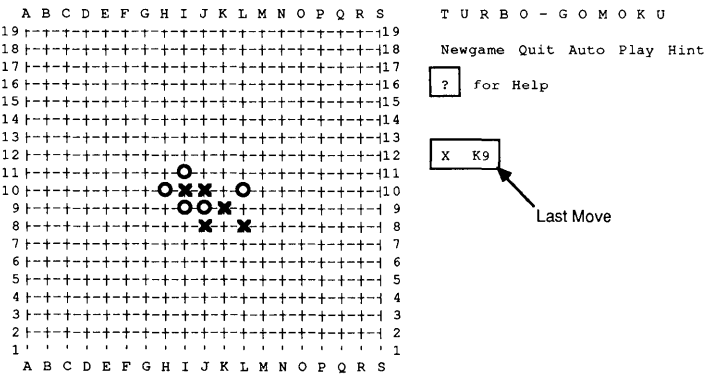


Figure 1-1. Turbo Go-Moku Screen

How to Play Turbo Go-Moku

On your distribution diskette is a file called GO-MOKU.COM which will let you try your hand at Go-Moku. When you run GO-MOKU.COM (by typing GO-MOKU on the DOS command line), the program displays a playing board containing 19 × 19 squares, labeled from A1 to S19. A small cursor appears in the center of the board.

To begin the game, press [↵] or the space bar after your piece is situated where you want it. (Traditionally, the first player places his or her piece on the center intersection, because this leaves plenty of room for playing.) If you'd rather have the program make the first move (note that the player who moves first has quite an advantage), press [P] (for play). You and the computer alternate moves from then on.

To make a move, use the arrow keys to move the cursor to the intersection where you want to place your piece, and then press [↵] or the space bar. In addition, the [PGUP], [PGDOWN], [HOME] and [END] keys move the cursor diagonally. You can also use the following commands:

Newgame	Starts a *New game.*
Quit	*Quit*s the program and returns to DOS.
Play	The program makes a move. Use this at the beginning of the game to let the program go first, or use it during the game to change sides with the computer.
Hint	Gives you a *Hint* about where to move.
Auto	The program *Auto*matically plays the rest of the game against itself.
?	On-line help about whatever you're doing at the moment. Scroll through the help text, then press [ESC] to resume play.

To enter a command, press the key of the first letter in the command name [N], [Q], [A], [P], [H] or [?].

It's a good idea to try to set up two different rows of three pieces each during the beginning of the game. The program will try to block every row of three that you arrange, but it is possible to win by setting up two of these threats simultaneously. The player who first gets four pieces in a row with *both* ends open will (if vigilant!) automatically win, because the program cannot block two places with one move.

That's all there is to playing Turbo Go-Moku. If you're interested in learning how the game is designed and how to make modifications to it, turn to Chapter 5.

Chapter 2
PLAYING CHESS

According to an Indian legend, chess was originated by a philosopher named Sassa. Sassa invented chess for his King, Shahram, in an attempt to devise a game that would rival backgammon in popularity. The King was so pleased by the invention that he offered to give Sassa anything he desired. Sassa asked for a seemingly modest reward—an arrangement of corn on the chessboard. He wanted one kernel to be placed on the first square, two kernels on the second square, four on the third square, and so on, doubling the number of kernels on each successive square until all the squares were covered. There is no record of what took place when the King discovered the true nature of this request, but the total amount of corn required to fill the entire chessboard would have been an astronomical 18,446,744,073,709,551,615 kernels!

Legend aside, it does appear that chess was actually invented in North India sometime around the year 500 A.D. The game was apparently supposed to symbolize two Indian armies going into battle. The chess pieces (King, Minister, Elephant, Horse, Chariot, and Foot Soldier) were realistic representatives of the members of the Indian army; their respective moves depicted both the importance of that role and the type of action it performed. For example, the foot soldier of the Indian army filled a weak, menial role—he had to trudge forward and attempt to kill or capture enemy soldiers; this was reflected in the game. The more cautious, circumspect movements of the King and Minister are also represented realistically. The actual method of winning was also portrayed accurately—while it was preferable to kill or capture the king, it was equally effective (although not as glorious a victory) to destroy the army.

Over the next six centuries, chess spread throughout Asia. It made its way to Persia, and then on to China, Korea and Japan. From the Middle East and Northern Africa, it spread to Europe, where the game entered the mainstream culture for the first time. Now, almost 1500 years after its invention, chess continues to fascinate people all over the world.

Popular games have always inspired new twists or variations; the intricacy of chess led people to dream of powerful machines that could play the game (see Appendix A for the history of chess machines). Such machines have intrigued people for centuries, but it is only during the last twenty years that they have become sophisticated

enough to challenge a human chess champion. In fact, computer chess is probably the *only* area of artificial intelligence in which the computer can outdo human experts.

How to Play Turbo Chess

You should know how to play chess to get the most out of Turbo Chess. If you don't know how to play, find a beginner's instruction book at your library or bookstore. Consult Appendix D for a list of recommended books. For your convenience, the rules of chess are given in Appendix B.

The file CHESS.COM on your diskette contains the compiled CHESS program file (the main source program is CHESS.PAS). Another file, OPENING.LIB, contains a library of chess openings that the program uses in the early game. Copy these two files to your working copy and then put the distribution diskette away. (Make sure you place the OPENING.LIB file in the same drive and directory as the CHESS.COM file. If OPENING.LIB is not found when the chess program is loading, an error message will be displayed before the game begins.) A script of the games you play will be stored in a text file called CHESS.

When you run CHESS.COM (by typing CHESS on the DOS command line), the program sets up the chessboard and presents you with a control panel to the right of the board.

Most chess openings are routine. Chess masters, in fact, memorize openings and responses. We've supplied Turbo Chess with a library of traditional openings and responses in a file called OPENING.LIB. Until you see the analysis information on the screen (depth, value, etc.), you'll know that Turbo Chess is playing from its opening library.

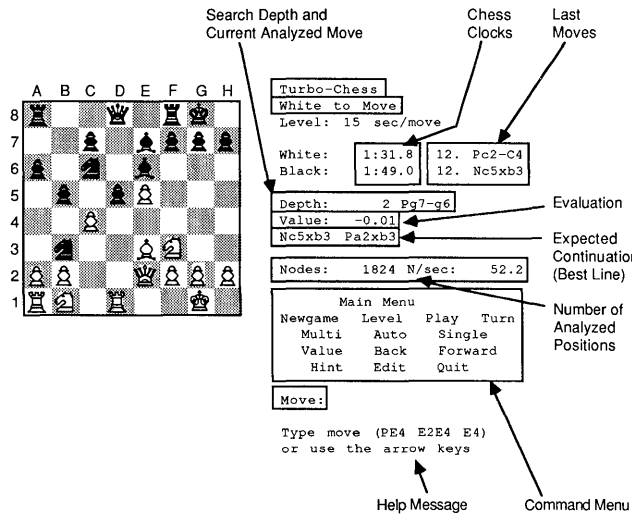


Figure 2-1. Turbo Chess Screen

Making Moves

There are three ways to move a chess piece in Turbo Chess: by use of the arrow keys, by algebraic notation, or by short notation. Turbo Chess does not use traditional chess notation (*KN-KB3*) because each position on the chessboard must be a discrete identifiable location in order for the program to know where each piece is.

The easiest way to make moves is to use the arrow keys and the space bar to point to and move the pieces. To use the arrow keys, press any of the four arrow keys [↑], [↓], [←] and [→] on the numeric keypad (make sure the [NUMLOCK] key is off). Four blinking arrows will appear in the middle of the board. Use the arrow keys to point to the piece you wish to move and then press the space bar. Now, move the blinking arrows to the square of your choice and press the space bar again. Additionally, the [HOME], [PGUP], [END] and [PGDN] keys move the arrows diagonally.

For instance, to move white's p-k4, do the following:

1. Press the [↓] key three times.
2. Press the space bar.
3. Press the [↑] key two times.
4. Press the space bar.

- *Half Move or Ply*—one move by one player.
- *Full Move*—one turn each for both players.

While using the arrow keys to make moves is easy, you might prefer to use algebraic or short notation. In algebraic notation, each square is labeled, from *A1* to *H8*. You move by entering the square you are moving from followed by the square you are moving to. With short notation, you specify the piece to move and the destination, e.g., *NF3*. You can also specify a move by typing in the destination square (legal only for pawn moves). The program makes its move after you make yours. To make the program move first, type [P] for play.

Non-queen pawn promotion is seldom used in practice, but including it in chess code makes for a far more sophisticated and flexible program. Like *en passant*, non-queen promotion takes time to implement as a "special case." Perhaps a satisfactory chess program could be designed more simply without these moves, but the resulting game would arguably no longer be chess.

Instead of entering a move when prompted, you can enter one of the other commands listed in the Quick Reference Guide on pages 14 and 15. Enter a command by typing the first letter of the name (in either upper or lowercase), followed by [↵]. For example, for Newgame, type *n*, followed by [↵]. Each of the commands is described in detail beginning on page 13.

Captures are entered just like any other move.

En Passant captures are handled by moving the pawn to the empty destination square; the captured pawn is removed automatically. An *en passant* capture is a special pawn move that the program sometimes uses. Many novice chess players don't know of its existence. But be on guard: Turbo Chess *does*.

Castling is accomplished by entering the king move only (e.g., *KG1*).

Pawn promotion can be entered in two ways. If you use algebraic notation, the program automatically gives you a queen. If you use the short notation, you should enter the desired piece type as the moving piece (for example *QE8* if you want a queen). The program generally chooses a queen for itself, but in some situations it may surprise you by taking a different piece.

The Opening Library

The opening library is a binary file named OPENING.LIB on the distribution diskette. During an opening situation, the number of moves a player can make is almost infinite, and the difference in value between them almost infinitesimal. We've supplied Turbo Chess with a library of traditional openings and responses. You will know that Turbo Chess is using this library when it displays the message "Using opening library" in the evaluation panel. When it has left the opening library, Turbo Chess will display its evaluation information (depth, value, etc.) in the evaluation panel.

How Turbo Chess Handles Errors, Checkmates, and Draw Games

If you try to make an illegal or impossible move, or try to cheat, Turbo Chess notifies you of the error and prompts you for another move. You cannot enter an illegal move in Turbo Chess.

The program also displays a message near the bottom of the control panel under the following conditions:

- If the program puts you in check
- If you put the program in check
- If the program resigns
- If the program finds a forced mate (with number of moves to achieve mate)
- If the game is a draw

- Draw by stalemate: the player whose turn it is to move cannot do so without entering check (illegal).
- Draw by third repetition: the same position appears three times with the same player having the move each of the three times.
- Draw by 50 moves: a maximum of 50 moves can take place since last capture or pawn move.

The program recognizes a draw by a *stalemate*, *third repetition* and the *50-move rule* (see Appendix E for a definition of these terms). The program will resign when it is heavily behind in *material* (number and value of remaining chess pieces), but it will *not* resign if it is being mated by you. When the program resigns or the game is a draw, you can always choose to ignore this by making the next move, thus forcing the program to continue playing move by move.

While the program is analyzing, it displays the best line found so far, as well as the evaluation of the position, the search depth in half moves, and the move that is currently being analyzed. The *best line* is the series of moves that the program expects the game will follow. A *full move* is one turn for each player, while a *half move* or a *ply* is a single move made by one player. The *search depth* is the number of half moves that the program has looked ahead. The value indicator

on the control panel shows how the program rates its own position in the game. It measures the evaluation in numbers of pawns (for example, +3.00 means that the program is three pawns ahead). A positive number indicates that the program believes it has the advantage; a negative number means the advantage is yours.

- *Best Line*—the series of moves that the program expects the game will follow.
- *Search Depth*—the number of half moves that the program has looked ahead.

During the search for a move, the program tells you the total number of analyzed *nodes*, as well as the number of analyzed nodes per second. A *node* is the point in computer logic where a number of potential moves diverge. The number of analyzed nodes is equivalent to the number of positions the program has analyzed.

The program also creates a listing of the moves of the game. This listing is stored on the disk in a file called CHESS. Any number of games can be stored in this file; however, you should rename the CHESS file if you wish to *permanently* save any games, because the file will be overwritten the next time you run CHESS.COM. If you want a hard copy of the game you just played, you can print out the CHESS file.

Sample Chess File Listing

No	Player	Program	Hint	Value	Level	Nodes	Time
1.	Pe2-e4	Pc7-c5	( )	0.00	0: 0	0	0.0
2.	Qdl-h5	Ng8-f6	(Qh5-e5)	0.04	3:19	1250	34.0
3.	Qh5-f5	Pd7-d5	(Bfl-b5)	0.05	2:12	997	33.0

This is a sample listing of the information Turbo Chess stores in the CHESS file. The CHESS file is always stored in the current logged directory on the logged drive—not necessarily on the games disk.

Turbo Chess Commands

Enter the letter of the command that is highlighted on the menu followed by [↵] (for example, to **P**lay, enter [P] [↵]. Also be aware that some letters are used twice (although in different menus) to indicate different actions. Thus, entering an [L] from the main menu will get you the **L**evel menu, but if you are in the **E**dit menu, pressing [L] will load a previously saved board from disk. Consult the Quick Reference Guide on pages 14 and 15 for an easy-to-follow flowchart and summary of the commands.

Newgame

To terminate the current game and start another, use the **N**ewgame command when prompted for a move.

Quit

The **Q**uit command stops the program and returns you to DOS.

Chart of Turbo Chess Commands

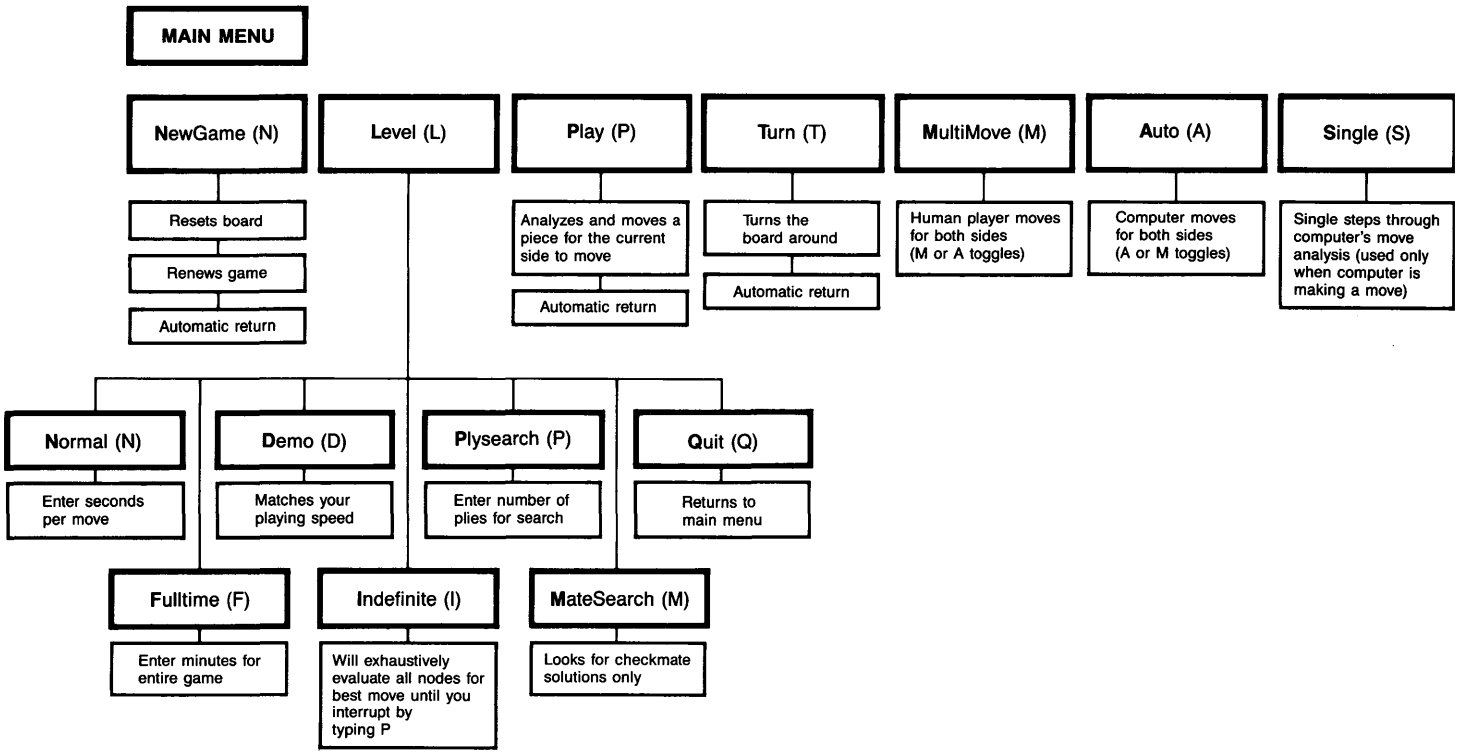
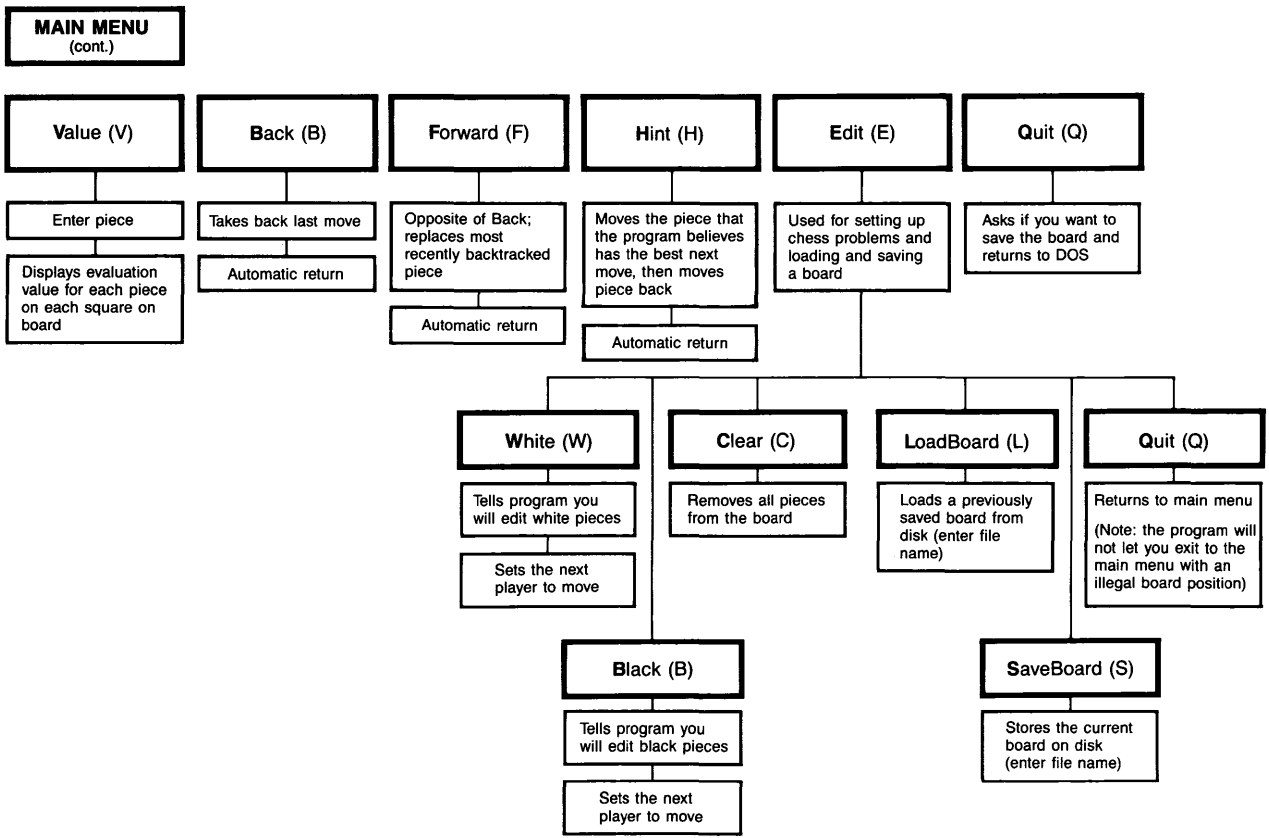


Chart of Turbo Chess Commands continued



Level

Turbo Chess takes, on average, 15 seconds to make a move. If you want to play at a different pace, you can set the level *before* you start the game using the **L**evel command. The Level menu is then displayed. There are six different ways to set the levels: **N**ormal, **F**ull time, **D**emo, **I**nfinite, **P**ly search and **M**ate search.

Level Menu Options

Normal	Enter seconds per move
Full time	Enter minutes for whole game
Demo	Plays at the same speed as you
Infinite	Analyzes until you interrupt
Ply search	Enter number of plies
Mate search	Looks for checkmate solutions
Quit	Returns to main menu

- For **N**ormal level, specify the average response time per move in seconds. The program will maintain a time budget, so that it makes a certain number of moves within the time you specify. The program uses more time for complex moves and less for simple ones, but it averages out the time over a number of moves. If it uses a lot of time in its opening moves (i.e., if it doesn't use the opening library), it will play more quickly later on. There is a time control after 40 moves and after each succeeding 20 moves. Thus, if you give the program 180 seconds (3 minutes) per move, it will play 40 moves in 2 hours, and then 20 moves per 1 hour the rest of the game.
- For **F**ull time level, specify the total time for the whole game in minutes. The program budgets its time as mentioned above but it will play the entire game within the specified time limit. Thus, if you want to play "blitz chess," give the program a total of 5 minutes (and if you're going to play fair, don't use more than 5 minutes yourself!).
- **D**emo level plays at the same speed as you do, regardless of whether you play quickly or slowly. This level is used when you want a nice, quiet home-style chess game with no concern about time.
- **I**nfinite level exhaustively analyzes each possible move until you terminate the search by pressing [P]. In other modes the program uses various algorithms to arrive at a move decision within a reasonable amount of time. You may use infinite level when you are playing chess by mail and can afford the time to let the program grind away undisturbed, or when you want to thoroughly analyze

a possible move. **Note:** Even at infinite level the program will use the opening library of moves, so to get an "infinite" analysis of traditional openings, temporarily move OPENING.LIB out of the current directory.

- **P**ly search level analyzes to a fixed search depth, measured in plies (half moves). This comes in handy when you want to experiment after changing the program, or when you want the program to play at a high level.
- **M**ate search looks for checkmate solutions. The program first tries to find a mate in 1 move, then a mate in 2, 3, 4, 5 . . . moves. When the program finds a definite checkmate solution, it plays it. You can then enter a defensive move, after which the program will complete the mate. A good way to make use of Mate search is to load one of the ready-made chess problem files on your diskette (MATE.01, MATE.02...MATE10). Set the level to Mate search, quit level mode, and then press [P] to play. Or, you could set up a chess problem from one of the popular chess columns in most newspapers and see what Turbo Chess comes up with for a solution.

Clocks

The program has two internal chess clocks, one for each player. When you change the level during the game, the chess clocks are automatically reset. For normal level, the clocks are set to the average time per move, multiplied by the number of played moves. For other levels, the clocks are set to zero. Thus, if you play in a tournament in which the time limit is 2 hours for 40 moves, plus 30 minutes for the rest of the game, you should start by giving the program 180 seconds per move, and then after 2 hours give the program 30 minutes for the rest of the game. A limit of 180 seconds results in 2 hours for 40 moves (3 minutes x 40 moves). Setting total time for 30 minutes at that point gives a 2½ hour tournament match.

Changing Sides and Terminating Search

When you enter [P] (for play), the program makes a move. Thus, if you start the game with this command, the program will make a move for white, and you will then play the black pieces. If you also enter the **T**urn command, the program will turn the board around, so that the black pieces are at the bottom of the screen.

If you enter the **P**lay command while the program is "thinking," it immediately terminates the search and makes the move which at that moment it considers the best. You can do this to speed up moves in Autoplay mode. To prohibit the program from making ridiculous moves, you are not allowed to terminate the search until the program has finished a search one half-move (one ply) ahead.

Taking Back Bad Moves

If you make a bad move—or any move, for that matter—you can take it back with the **B**ack command. The **B**ack command takes back the most recent one half-move. For instance, if the program captures your queen, you can recover it by taking back two half moves, one for the program and one for yourself. You can take back as many moves as you like, all the way to the beginning of the game. The **B**ack command enables you to evaluate your previous moves and thus strengthen your game. Afterwards, you can enter **F**orward to march forward through the moves of the game again, and return to where you were before you started **B**ack.

Need a Hint?

Enter **H**int, and Turbo Chess "advises" you by playing the best move on the board. You can use the **H**int command both when it is your turn to move and while the program is analyzing.

Changing Colors and Setting Up Positions (The Chess Board Editor)

When you start experimenting with the program, you will probably want to set up a particular position or chess problem. You can do this with the editor, which is invoked with the **E**dit command. The edit menu will then appear. Place white and black pieces on the board by entering the piece type followed by the square name (for example *QD8* to place a queen on D8). You can use the arrow keys to specify the destination position and edit the board. Chess problems can be set up on a clear board, or you can enter the editor in the middle of your game and add or subtract pieces. You can clear the whole board with the **C**lear command. You can save chess positions you set up with the **S**aveBoard command, or load a position with the **L**oadBoard command. Turbo Chess comes with a number of chess problems already laid out; these programs are contained in the files MATE.001, MATE.002..MATE.010.

You can change which color will move next with the **W**hite and **B**lack commands. Remove a piece from the board by entering a space, followed by the square name. Leave the editor with the **Q**uit command. You must have a legal number and configuration of pieces on the board in order to quit the editor. When you leave the editor, the color next to move will be the same color as the last entered piece.

Setting Up a Board Using a Text Editor

The chess program allows you to save boards and reload them from disk files. Therefore, once you know the format of a saved board, you can easily set up a board position by using a text editor to enter the pieces and positions:

```
WHITE to Play    You can put comments here. Date match
WHITE=Program    played: 22 OCT 1985 with Koltanowski.
WK H2
WP G3
BK C4
  .
  .
  .
```

The first line indicates the color of the next piece to be played. The second line indicates which side the computer is playing. The lines that follow indicate which pieces are on the board; they are in the following format:

```
[Color: W or B][Piece: P,N,B,R,Q,K] [File: A-H][Rank: 1-8]
```

For an example of a board saved to disk, please refer to one of the MATE files included on the diskette.

Note that you can easily set up a game using the EDIT selection on the chess program's main menu (see section above).

Auto

Auto causes the program to play against itself; this might result in a better game than if you play against the computer, and can thus help you improve your own game. When a game is finished, the program starts a new one. Stop the sequence of games by entering **A**uto again, or by entering [M] for **M**ultiMove.

MultiMove

The **M**ultiMove command permits you to enter moves for both you and the program (or play with another human friend). This feature is useful when you want to set up a particular opening position, or when you otherwise experiment with the program. To return the program to regular playing mode, use the **M**ultiMove or **A**uto commands.

SingleStep

The **S**ingleStep command is used primarily for debugging the program or to take a look at how the program "thinks." In **S**ingleStep mode, the program displays its analysis on the screen move by move, one move each time you press [↵]. To turn off **S**ingleStep mode, press [Q] before you press [↵].

Value Display

The **V**alue command is also used for debugging. Enter a color and a piece type (for example, *WQ* for white queen), and the program will print its evaluation of all moves for that piece on each of the 64

squares. Leave the value mode with the **Q**uit command. The evaluation function is described in detail in Chapter 6, Chess Program Design.

You now know how to use the Turbo Chess program. We think you'll find it plays a good game. For a guided tour of the source code, turn to Chapter 6. For a look into the fascinating history of machine chess, turn to Appendix A.

Chapter 3
PLAYING BRIDGE

Bridge is one of the most popular card games in Europe and North America. In the United States it's even more popular than chess, probably because of the social nature of the game and because it offers an ideal combination of fun and intellectual stimulation.

You should already know how to play bridge to get the most out of Turbo Bridge. If you don't know how to play, find a beginner's instruction book at your library or bookstore. Consult Appendix D for a list of recommended books. For your convenience, a brief description of basic bridge rules and strategy is given in Appendix C.

How to Play Turbo Bridge

The file BRIDGE.COM on your diskette contains the compiled BRIDGE program file. When you run BRIDGE.COM (by typing *BRIDGE* on the DOS command line), the *options* menu will appear.

Options Menu

The options menu lets you select how you want to play. You can change any of the default settings described below or just press [P] to begin playing. To move the cursor (the two blinking arrows) around within the options menu, use the [←], [→], [↑] and [↓] keys on the numeric keypad (make sure [NUMLOCK] is off).

(**Note** that upper and lowercase letters are treated the same in Turbo Bridge; a capital P is equivalent to a lowercase p, and so on. Just use whichever is easier for you.)

The default settings given in the opening menu are listed below.

1. **Select hands you wish to Play**: NORTH EAST **SOUTH** WEST
   [default = SOUTH]
   To select the hand(s) you wish to play, position the cursor over the desired hand and press the space bar. All hands that you select to play will appear on the screen in reverse video. The program will play any hands you don't select.

2. **Display all 4 hands**?
   [default = NO (only your hands are displayed)]
   Do you want the program to display all four hands or just the hand(s) you are playing? (Use the arrow keys to position the cursor over the answer and hit [Y] for YES or [N] for NO, or simply press the space bar to toggle between YES or NO.)

3. **Should the program cheat and look at your cards?**
[default = NO]
Do you want the program to cheat by looking at all hands on the table? Use the arrow keys to position the cursor over the answer and press the space bar to toggle between YES and NO. (Note: as with human players, the program will play a stronger game of bridge if you let it peek at your cards.)

4. **Play**
When you're ready to start playing, press [P] or position the cursor over the Play box and press the space bar. The bridge screen and *bid menu* (described in the next section) will then be displayed. The bridge screen appears as a square bridge table; the bids and played cards for each trick will be placed on the table.

Turbo Bridge uses the same bridge layout convention as most books and newspapers: **E**ast-**W**est, **N**orth-**S**outh as partners (abbreviated as **E**, **W**, **N**, **S**), with **S**outh at the bottom of the screen.

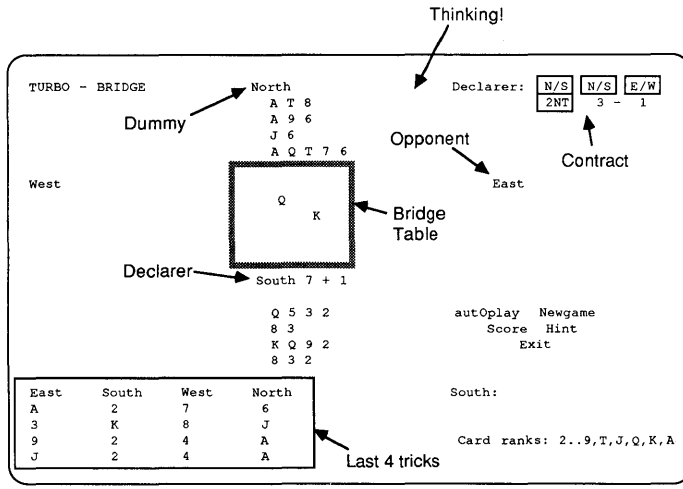


Figure 3-1. Turbo Bridge Screen During Play

Bidding

To prompt you for your bid option, the hand you are playing will appear on the screen (e.g., if you are playing South and it is your turn, the prompt "South" will appear).

The program automatically bids and plays its own hands. The bids and the played cards are both shown on the bridge table and in the information window in the lower left part of the screen.

To enter a bid, type the bid level (**1–7**) and the suit. The suits are entered as **C,D,H,S** and **N** for **C**lub, **D**iamond, **H**eart, **S**pade and **N**otrump. Thus, one club is entered as **1C** (or **1c**) and 7 notrump is entered as **7N** (or 7n).

If you make a syntactically correct bid that is illegal, for example, opening the bidding with **D**ouble, the computer will ring a bell and tell you that you have made an invalid bid.

The bid menu options are described below. To select an option, press the key indicated by the highlighted letter on your screen.

- [**D**]ouble. Press [D] to double the bonus for making a contract and the penalties for defeating the contract.
- [**R**]edouble. If your opponent has doubled your bid, you can, if you wish, press [R] to redouble the bid.
- [**P**]ass. To pass during the bidding, press [P].

Four other options are also available:

- [**C**]lear Bids. Press [C] to clear all bids without re-dealing the cards. This is useful if you make a blunder when bidding and want to try again.
- [**N**]ewDeal. If you don't like the hand you've been dealt, you can tell the computer to re-deal the cards by simply pressing [N]. This option also clears the bids. You will be given the option of returning to the default menu before new cards are dealt.
- [**S**]core. To view the current score, press [S] from within the bid or play menu (the play menu is discussed on page 27). The menu will be replaced by the score card. The score is accumulated until one of the teams wins a rubber (two out of three games). To return to the appropriate menu after viewing the score, press any key.
- e[**X**]it. Press [X] to end the game. Pertinent information about the game you just played will be stored in a text file named "BRIDGE."

Bidding Quick Reference

Bidding

1 club	1C
1 diamond	1D
1 heart	1H
1 spade	1S
1 notrump	1N
Double	D
Redouble	R
Pass	P

Other Options from Bid Menu

Clear bids	C
New deal	N
Current score	S
eXit to DOS	X

(Remember that upper and lowercase letters can be used interchangeably; for example, 1C and 1c both indicate 1 club.)

Bidding ends when a bid is followed by three consecutive passes.

The Bidding System

The program uses a homemade bidding system called *Simplified Goren with Modifications*. As with any computer program, we must sometimes restrict input to a set of factors we can handle. We'll talk more about this in the Program Design section of this book.

To get the most out of Turbo Bridge, you should conform to the Simplified Goren system detailed below. For more information, see Goren's books (some of which are listed in Appendix D, "Suggested Reading").

High-card points are counted as usual (ace 4, king 3, queen 2 and jack 1). Distribution points are 3 for void suit, 2 for singleton and 1 for doubleton. The distribution points do not count in NT bids.

Below you can see how many combined points you and your partner should have in order to bid game or slam:

4 in Suit	26 points	3 NT	26 high card points
5 in Suit	29 points		
6 in Suit	33 points	6 NT	33 high card points
7 in Suit	37 points	7 NT	37 high card points

Table 3-1. Bidding Conventions

Bid	Example Bidding Sequence	Requirements For Bid
(Opening Bids)		
1 in suit	(1H)	13 to 23 points, 4 trumps
2 in suit	(2H)	24 or more points, 4 trump
3 or more in suit		0 to 11 high card points, 7 to 13 trumps
(Notrump Bids)		
1 suit, NT	(1H, 2C, 2 NT)	12 to 15 high card points
1 NT	(1 NT)	16 to 18 high card points
1 suit, NT jump	(1H, 2C, 3 NT)	19 to 21 high card points
2 NT	(2 NT)	22 to 24 high card points
3 NT	(3 NT)	25 to 27 high card points
(Response to Opening 1 in Suit)		
Pass	(1H, Pass)	4 to 5 points
2 in suit	(1H, 2H)	6 to 9 points, 4 trumps
3 in suit	(1H, 3H)	13 or more points, 4 trumps
1 higher suit	(1H, 1S)	6 or more points, 4 trumps
2 lower suit	(1H, 2C)	10 or more points, 4 trumps
1 NT	(1H, 1 NT)	6 to 9 points
2 NT	(1H, 2 NT)	12 to 15 points
(Response to 1 NT)		
Pass	(1 NT, Pass)	0 to 7 high card points
2 in suit	(1 NT, 2H)	0 to 5 high card points, 5 trumps
Stayman	(1 NT, 2C)	6 or more high card points
3 in suit	(1 NT, 3H)	10 or more points, 4 trumps
2 NT	(1 NT, 2 NT)	8 to 9 high card points
(Response to Stayman)		
2H or S	(1 NT, 2C, 2H)	4 trumps
2D	(1 NT, 2C, 2D)	Anything else
(Response to 2 in Suit)		
Not 2 NT	(2H, anything)	1 ace and 1 king
2 NT	(2H, 2 NT)	Anything else
(Other bids)		
Opponent overcall	(1H, 2C)	8 high card points, 5 trumps
Jump overcall	(1H, 3C)	0 to 11 high card points, 7 to 13 trumps
Stayman	(1 NT, 2C)	
Blackwood	(4 NT or 5 NT)	Slam interest
Double		*Always* natural
Redouble		*Always* natural
(Response to Blackwood)		
5C		0 aces
6C		0 kings
5D		1 ace
6D		1 king
5H		2 aces
6H		2 kings
5S		3 aces
6S		3 kings
5 NT		4 aces
6 NT		4 kings

All four-card suits are biddable (even 5-4-3-2). There is no weak club bid. When choosing between four-card suits, the lowest ranking suit is bid first. If a five-card suit is available, the highest ranking such suit is bid. Thus, with a 3-4-4-2 distribution you open in diamonds, with a 4-1-4-4 distribution you open in clubs, and with a 5-5-2-1 distribution you open in spades.

With notrump distribution (4-3-3-3, 4-4-3-2 and sometimes 5-3-3-2) you always bid NT, either immediately or in the second bid. Opening one in a suit followed by NT at the lowest possible level means 12–15 high card points (hp), while opening one in a suit followed by jump in NT means 19–21 hp.

Table Talk With a Computer? Bidding Strategy

With the modified Goren system you can exchange a lot of information without bidding very high. For example, opening one spade means either you hold five spades or the very rare 4-3-3-3 distribution. With 3-4-4-2 distribution and 12-14 hp you open one diamond. If your partner has four hearts, s/he will bid them (unless of course s/he also has a five-card suit). Thus, if s/he does not bid one heart, you can assume that s/he does not have four hearts. So unless s/he bids hearts or diamonds, you will show your NT distribution in the second bid. If s/he bids one spade you bid one NT, and if s/he bids one NT or two clubs you bid two NT.

One NT in response to opening one in a suit does not promise anything about the distribution. It just means that you cannot bid any of your four-card suits. Thus one NT in response to opening one diamond means no four-card suits in hearts or spades.

Bidding two in a suit (other than clubs) in response to opening one NT is a very weak bid, which means that the player thinks two in the suit is better than one NT. The partner should always pass to this bid. Two clubs in response to one NT (or three clubs in response to two NT) is a special bid called the *Stayman convention*. It shows nothing about the club suit, but asks the partner to show four-card suits in hearts or spades. If the partner has any, s/he bids it, otherwise s/he bids diamonds. If the partner has four cards in both hearts and spades s/he bids hearts.

Opening two in a suit is a strong bid, which always forces to game. Two NT in response is an "artificial bid." It shows a weak hand. Any response other than two NT means slam interest and suggests at least an ace and a king.

The first four NT bid in a game is a special bid called the *Blackwood convention*. It is used to bid slams, and it asks the partner how many aces s/he has. The partner bids five clubs with no aces, five diamonds with one ace, etc. up to five NT with four aces. The first player can then ask for the number of kings by bidding five NT (this cannot be done if the partner has shown four aces).

Notice that doubles and redoubles are always natural. An overcall bid requires a five-card suit, and a shutout bid requires a seven-card suit.

Trick or Treat: Playing the Cards

The play begins when the bidding is finished. The declarer is the partner of the pair winning the contract to first bid the contract suit. The dummy is the declarer's partner. The player to the left of the dummy (clockwise) plays the opening lead. Then the dummy's cards are revealed. After the opening lead, the screen picture is turned around, so that the declarer is placed at the bottom and the dummy at the top of the screen.

The declaring side, the contract bid, and the current number of tricks won by each side are displayed in the upper right hand corner of the screen. When the computer is playing the current hand, this is indicated on the screen (e.g., "West to Play"). When it is your turn, the play menu will come up and you will be prompted with the hand you are to play (e.g., South). (If you are in AutoPlay mode, the message "AutoPlay" is displayed.)

Enter a card to play simply by entering the rank and suit. The values are entered as **A,K,Q,J,T** for **A**ce, **K**ing, **Q**ueen, **J**ack, **T**en and **9–2** for the low cards. Thus **AS** means ace of spades, **TH** means 10 of hearts and **2C** means 2 of clubs. If just one suit is being played, just enter the value. If clubs have been led, **A** means ace of clubs. You must follow suit if possible; the program strictly enforces this rule. If only one card can be played, the program will play that card automatically without asking you to enter a card. If you press [↵] the program will play the lowest card that can be legally played. Trumps are considered to have higher value than non-trump cards. If two cards have the same value, the card in the lowest suit will be played. For example, if clubs are led, press [↵] to play the lowest club. If you have no clubs, press [↵] to discard the card with lowest value.

In addition to playing the cards, there are four other options available from the play menu:

- [**S**]core. Press [S] to view the correct game score.
- Aut[**o**]Play. Press [O] to tell the computer to play out the hand for both bridge teams. This is useful if you have made the contract and you are not concerned with the outcome of the game.
- [**H**]int. Press [H] to ask for a hint. The computer decides what card you should play and displays it on the command line. To play the suggested card, press [↵]. To clear the hint, press the [BACKSPACE] key.
- [**N**]ewGame. Press [N] to cancel the current game; the score is unaffected. This is useful when you are learning how to bid and are unconcerned about playing the cards.

- e[**X**]it. Press [X] to end the game. Pertinent information about the game you just played will be stored in a text file named "BRIDGE."

Play Quick Reference

ace of spades	AS	
king of hearts	KH	
queen of diamonds	QD	
jack of clubs	JC	
jack of lead suit	J[↵]	(if suit has been led, you don't have to specify it)
ten of spades	TS	
9 of hearts	9H	
2 of clubs	2C	
Computer selects low card of current suit	[↵]	

Other Options from Play Menu

Score	S
AutoPlay	O
NewGame	N
eXit to DOS	X

If you play an invalid card, the computer will ring a bell and tell you that you have made an invalid play.

The bridge table displays the played cards for each trick. When a trick is completed, the number of won tricks will be updated and the cards will remain on the table for your analysis until a key (any key) is pressed.

When the game is finished, the score card is displayed with the game results. The score card remains on the screen until any key is pressed. Some typical game result comments are: "N/S made contract with one overtrick," which means that the declarer got an extra trick, and "E/W down 1," which means that the contract was missed by one trick.

The score will be displayed until you hit any key to continue to the next game. You will then be asked if you want to reset the playing options before resuming play.

The program creates a listing of all the deals. This listing is stored on the disk in a file called BRIDGE. For a hard copy of the games you have played, you can print out this file. This is useful when you experiment with the program.

Section II

INTO THE SOURCE CODE

Chapter 4
ANATOMY OF A GAME

This chapter introduces you to computer game design concepts that are taken for granted throughout Section II of this manual. In particular, three fundamental parts of any strategic computer game are discussed: data structure, evaluation, and user interface.

Types of Computer Games

Three different game genres have emerged during the relatively brief history of the game-playing computer:

- *Strategy Games*—board games and card games, or games played on a (sometimes imaginary) grid. All three games in the Turbo GameWorks package fit this category. These games are *strategic* because of the superiority of one position over another, calculated according to the rules of the game. Games like bridge and chess figure into the strategy category because players use an array of 52 cards with different values, or an array of 64 squares and pieces with different values.
- *Hand/Eye Games*—"shoot-'em-ups", driving games, and other video games that pit the speed and coordination of the human player against the processing power of the computer. These games generally use a highly detailed graphic screen display. Strategy here may be limited to hiding in the rocks or among the asteroids as you shoot. The displays for some of these games are extremely realistic. Turbo GameWorks does not deal with games of this type.
- *Adventure Games*—you assume the identity of a character (an array of values, again) in a story-like puzzle that is often represented textually as an underground maze of tunnels and rooms. Zork is one of the more well-known adventure games. The program translates your responses to different "events" into a better-or-worse position for solving the puzzle. Turbo GameWorks does not deal with role-playing games.

These game genres emphasize different aspects of computer game playing: positional strategy, sophisticated graphics programming or puzzle-solving. Under the skin, though, they are all much more alike than they are different. Strategy games, hand/eye games and adventure games all share common elements, although we'll be concentrating on strategic games in Turbo GameWorks.

Strategic games are a good place to start because people have been using board and card games for centuries; we are used to them. The procedures that display the board on the computer's screen are rel-

atively easy to understand, and the data representation of the board and pieces makes good sense.

For a hand/eye game, we'd have to take a lot of time to explain complex shape and animation routines that have little to do with the design of the game itself. In an adventure game, the data structure is more intricate, since part of the fun of these games is that the structure of the puzzle is not immediately obvious. In both hand/eye and role-playing games, these complexities can distract from the real heart of any game: the evaluation procedure that selects the program's move and interprets your response.

How to Recognize a Game When You See One

Every game has a data structure, a way to communicate with the user, and a procedure to evaluate which moves to make. These correspond to the aspects of a game as it is played; different games emphasize different parts of a program. An emphasized data structure may reflect position or strategy; a high amount of communication pushes the hand/eye coordination aspect; and puzzle-solving emphasizes the evaluation of the user's moves and the program's selection of its own moves. They are the same ingredients in all three instances—just mixed differently for the different types of games.

Data Structure

The challenge to a game designer (or any programmer) is to create a data structure that accurately represents the field of play, moves, and program status, and that helps the program run smoothly with a minimum of code.

The Go-Moku program, for example, builds up the variable *Board* from three simple elements:

```
const
   N = 19;
 type
   Boardtype = (Empty, Cross, Nought);
   Indextype = 1..N;
 var
   Board : array[Indextype,Indextype] of Boardtype;
```

Translated, this means the board is a 19×19 grid, where each square in the grid may hold the value called "Empty," "Cross" (X), or "Nought" (O).

It takes a great deal of thought to design good data structures. In Turbo Bridge, for instance, we'll see that we have two problems to represent: the cards that the players actually have, and the cards they *claim* they have through their bids.

Data structure is the logical organization of information about the board, playing pieces, the moves, and winning or losing. For example, in a chess game we might have a record variable for each square on a chess board, like this:

```
Square = record
           Piece : boolean;
           King  : boolean;
         end;
```

Then we could create an array [1..64] of type *Square*, and be able to track whether any particular square holds a piece, and if that piece is a king.

Depending on whether our program can make efficient use of this particular data structure, it might be a good way to organize a chess game.

Evaluation

The heart of any strategic computer game is the *evaluation function.* This is the part of the program that examines possible moves, weighs them against a set of rules for good and bad moves, and selects from among them.

The ability to look ahead, or *search,* is crucial to strategic computer games. Three different search techniques are used in Turbo GameWorks.

- In the Go-Moku program, we use *no search,* and instead rely solely upon the evaluation function. This method is simple, but the evaluation function must be very reliable for it to be effective.
- In the Chess program, we use a *brute force* search, which means that we analyze all possible moves to a fixed depth.
- In the Bridge program, a brute force search would be too time-consuming. We therefore use a *selective search,* which means that we analyze only some of the possible moves. Deciding which moves to analyze can be difficult; designing a reliable selective search can thus be a challenge. The Bridge program also shows how to handle unknown information (for example, when each player knows his or her own cards but not those of the opponent.)

User Interface

User interface is an element that every game designer must consider to make a game interesting to play. Admittedly, you can write a chess game that prints its moves out line by line on a printer—but it's far more entertaining to draw a board on the computer screen and move the chess pieces from square to square as you watch. A good user

interface will allow the user to concentrate on the game rather than on communicating with the program.

The screen I/O in Turbo Gameworks is a compromise between the delightful and the expedient. You can display Turbo games on any IBM PC monitor, with either a monochrome or color card installed. Game pieces are represented as text characters instead of in graphics. This is especially apparent in the Turbo Chess game.

Because of the nature of strategy games and the importance of the evaluator functions, Turbo Gameworks spends a minimum amount of time (and code) on I/O. In every Turbo game program, however, I/O is handled in an easily identifiable section of the program. If you want to modify Turbo Chess, for instance, to display graphic chess pieces, go right ahead. Turbo Graphix Toolbox may help you there.

And They're Off!

A game may be simple in concept and complicated in programming and execution; or it may be complicated in concept but elegantly simple in data structure and evaluation function. For a person who enjoys discovering the internal logic of games, Section II of this manual should help you understand more about what makes Turbo Games tick. For the person who just wants to play—you can simply read Section I and start playing!

Chapter 5
GO-MOKU PROGRAM DESIGN

Let's take a look inside the Turbo Go-Moku program. Besides letting you in on our program design, this chapter will help you develop guidelines for creating your *own* Go-Moku game, and point you in the right direction if you'd like to modify the source code for Turbo Go-Moku.

We start by developing rules of thumb for program design: how to get our program to evaluate and make moves. The last part of the chapter takes the Turbo Go-Moku program apart module by module.

Go-Moku Rules of Thumb

1. Make 5 in a row if you can.
2. Stop your opponent from making 5 in a row.
3. If program has an open 3 and can make an open 4, *do it.*
4. If opponent has an open 3 and can make an open 4, *block it.*
5. If the program can make two different open 3's simultaneously, *do it.*
6. If opponent can make two different open 3's simultaneously, *block one.*

A computer program cannot *really* think on its own; you have to tell it exactly what to do. But you clearly cannot tell it what move to make from every possible board position; even in a deceptively simple game like Go-Moku, the first five moves can be made in 5,962,870,725,840 (that's *five trillion)* different ways! So if each series of five moves were to take up, for example, twenty bytes of storage, we would need a 100 gigabyte machine for our program—more than the storage capacity of a mainframe computer. Clearly, we must give the program some general rules of thumb that can be used in every possible situation. Deducing the rules of thumb is the first step in good computer game design.

What rules of thumb can we think of? To begin with, we know that the object of the game is to get five pieces in a row. Designing a program that checks for this condition is not a difficult task. Another rule is that if the opponent has four pieces in line, and the fifth grid intersection next to those four is empty, the program should place a piece on this open grid intersection. This prevents the opponent from winning on the next move. It appears that the first rule is the most important of the two, because it is better to win than to simply prevent the opponent from winning.

What else can we think of? After you have played the game a number of times, you will probably notice the importance of an *open 4*. An open 4 is four pieces in line, with empty grid intersections on both ends. No matter where your opponent places the next piece, a 5-in-line is inevitable on the following move. Just as powerful, but occurring less frequently, is the combination of two different 4s, each with one available empty grid intersection.

We can make two new rules here: if the program has an open 3 and can make an open 4, then it should do so. If the opponent has an open 3 and can make an open 4, the program should block it.

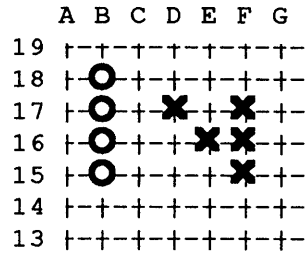


Figure 5-1. An Open 4 and Two Open 3s

Another strategic technique to force your opponent into a corner is to make two different open 3s. The opponent can only block one of them, and on your next move you can expand the other to an open 4. Thus, if the program can make two different open 3s, it should do so. If the opponent can make two different open 3s, at least one of them should be blocked.

Can you see a pattern forming? It is never possible to foresee every move in every game. However, a good game designer tries to *reduce* the possible moves into categories of moves, or rules, which can be applied to a variety of situations by the game program.

Basic Playing Strategy

Let's see if we can make this into our initial playing strategy. First, try to make a 5-in-line. Secondly, if the opponent can make a 5-in-line, block it. If none of these rules apply, try to make as many 4s as possible. We can begin to "grade" possible moves by giving them relative values. For example, an open 4 should count as two 4s, since it can be expanded in two ways (a blocked 4 that cannot be expanded to a 5 should not count). Next, try to block as many of the opponent's potential 4s as possible. Then try to make as many 3s as possible. We will count a completely open 3 as four 3s, since it can be expanded in four different ways. Next, try to block as many potential 3s as possible.

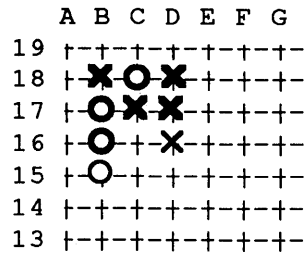


Figure 5-2. An X at D16 Makes Two Open 3s

Now that we've come up with this first strategy, let's think through our rules and see what can go wrong. Remember: the computer will only do what we tell it, and no more; it will play blindly by any rules we give it. Take a look at Figure 5-2, assuming it is 0's turn to move. According to our strategy, the program should expand its own 2 to a 3 by placing an 0 at *B15*. The opponent will then be likely to make two open 3s by placing an X on *D16*. Next, the program will place a 0 on *B14* and the opponent will block with an X on *B13*. The opponent then has two open 3s, and has won the game. What went wrong?

In order to make a single 3 ourselves, we let the opponent make two open 3s, and that was a bad idea. If the program has two moves that make two open 3s, we would like it to choose the move that also makes some 2s. It looks like our first strategy doesn't really work. Let's build on what we've learned and create another.

Evaluating Moves

Instead of using our rules of thumb in the same order every time we make a move, let's decide which rule is most appropriate at a given time, for a given move. What if we assign a value to each rule, rate each possible move according to those rules, add all the values together and pick the move with the highest value? We can award an increasing number of points for making a 2, 3, 4 and 5-in-line play. A 2-in-line rates 5 points; a 3 rates 25; a 4 rates 125; and a 5 rates 625 points. The points given for blocking the opponent's potential lines should be a bit lower— let's say 4, 20, 100 and 400 points respectively. What we've created is an *evaluator* that, given a move (and a board position), evaluates the value of that move.

Move	*Scoring Points* Making	Blocking
2-in-line	5 pts	4 pts
3-in-line	25 pts	20 pts
4-in-line	125 pts	100 pts
5-in-line (WINS)	625 pts	400 pts

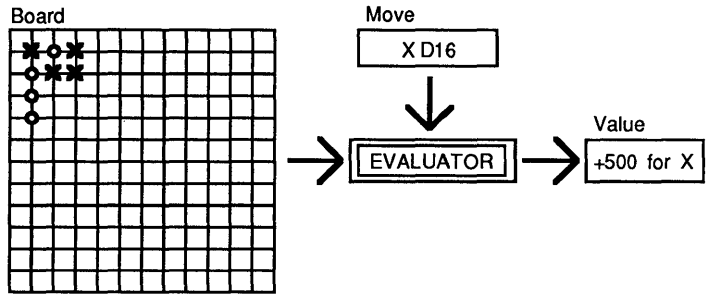


Figure 5-3. Go-Moku Evaluator

The evaluation strategy we've outlined here is one used by many expert systems. The Turbo Go-Moku program only plays a game; but one of the distinguishing characteristics of an expert system is that it can tell you which of its rules it is applying to a particular decision. When you are familiar with the program, you may want to modify it to be an expert: tell you which rule it is applying to each move.

The program now has several rules that are assigned different weights to help it choose its moves. To pick a move, the program assesses each possible move, using our rules of thumb to calculate a value for each move. This is called "evaluating the move." Then, it picks the move with the highest value. In short, we try all possibilities, calculate an evaluation for each, and pick the one with the highest score. All this is performed by a procedure called *FindMove* (described in greater detail later in this chapter). *FindMove* calculates the score for each unoccupied location on the board.

The method used by *FindMove* is employed by many computer game programs. Hans Berliner used it in a backgammon program that beat the world champion in a match *(Scientific American,* June 1980). It has also been used in bridge, go, and poker game programs. With small modifications, it is one of the key decision-making methods used in artificial intelligence research. Some expert systems evaluate rules using this method.

Problem Analysis

Let's review what we did in this section. We started out by analyzing the problem thoroughly, then set up a number of demands which our solution ought to meet. From the analysis and the demands, we reached an excellent solution—except that it didn't work. From this failure we got the idea for a much better solution, one that satisfied

Each position is part of up to 20 five-position lines. This is because a position can hold any of five spots along the line—beginning, middle or end—and sits along four possible line directions: up/down, right/left, and diagonals. ($5 \times 4 = 20$).

all the demands. This process is called *problem analysis*, and is the most important part of any software development project. Unfortunately, some programmers skip it entirely, and the result is a lot of excellent programs that solve the wrong problems.

The set of rules we've defined to make our decision is called an *algorithm*. The algorithm used by *FindMove* evaluates each possible move. There are a maximum of 361 possible moves if the playing board is empty (since $19 \times 19 = 361$ squares). (The *FindMove* procedure does not evaluate a position unless it is empty.) Each position on the board is part of up to twenty potential 5-in-lines. *FindMove* checks to see how a move would affect each of these 5-in-lines. To do this, it must evaluate each of the other four positions in each line. This evaluation results in up to $361 \times 20 \times 4 = 28{,}880$ checks to evaluate one move. This would probably take about three seconds on an IBM PC—not an unreasonable amount of time. Still, the faster we can make our program, the better it will be.

It's always the innermost loop that takes all the evaluation time. If you cut the time taken by the innermost loop, you can speed up the program accordingly.

When loops are nested, it is always the innermost loop that takes up most of the execution time. In this case, the innermost loop checks the four positions in a line to see how a new piece in the fifth position would affect the line. There are only two possibilities. If the line contains only the program's pieces, placing a new piece would expand the line. If the line contains only the opponent's pieces, placing a piece would block the line. Otherwise, the line is of no interest to the program's decision-making strategy.

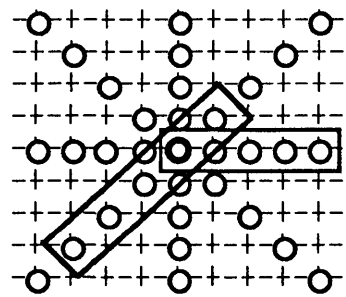


Figure 5-4. Each Position is Part of 20 Lines

Incremental Updating

Because of this, all our program needs to know is the number of O's and X's in the particular five-position line that contains the spot being evaluated. Instead of counting them again and again, why not keep the count in an array? There are 15×19 horizontal, 19×15 vertical and $2 \times 15 \times 15$ diagonal lines, giving a total of 1020 possible five-position lines on the board. Calculating the array from scratch requires five checks per line. Making a move affects only 20 of the 1020 lines; the numbers for the rest of the lines remain unchanged. Instead of calculating the entire array from scratch every move, why not just update it each time a move is made? This very useful method is called *incremental updating*, and is used in many programs for which speed is essential.

Even though incremental updating reduces the number of evaluations we must do (and speeds up the program!), we still need to do 20 calculations to evaluate each of the 361 possible moves; that makes 7220 calculations. What can we do to reduce that number?

Turbo Go-Moku avoids the use of labels; Turbo Chess and Bridge do not. Generally, it is bad programming style to use a lot of labels—they disorganize the code and make it harder to read. A few labels, however, may be permissible if they result in code that is more logical and easier to understand.

Each time we make a move, it only changes the values of the 32 grid intersections that belong to *1* of the 20 five-position lines attached to the square in this move. We can use incremental updating again to maintain a table showing the evaluation for each position. When we make a move, we change the numbers for the 20 possible five-position lines. For each of these lines we must change the evaluation for the four other positions in the line, each of which has its own set of 20 five-position lines. In total, the program has 100 calculations to make each time a human opponent makes a move, plus 361 calculations for each time the program moves. We have speeded up the program by 50 times using the trick of incremental updating—eliminating the unnecessary calculations. These tightened evaluations are far more sophisticated; the other way, evaluating every move, is called "brute force." Brute force is sometimes the only way to get a result, but you should always look for a more elegant alternative.

Go-Moku Program Structure

The Go-Moku program is contained on your distribution disk in the files named GO-MOKU.PAS and GO-HELP.INC. Use Turbo Pascal to modify and compile GO-MOKU, and use the Turbo Pascal editor to examine the source code. The first few lines tell you what the program does, who wrote it, and most important, *when it was last edited.* In the GO-MOKU.PAS program, the main body of the program takes care of communication with the user.

The main program consists of three parts (one for each label):

- the *initialization* part initializes variables
- the *command* part reads and performs commands (or moves)
- the *program move* part finds and makes a move for the program
- the *help* part is an on-line help system

Procedure *ResetGame* sets variables to their default values and resets the game. Next, a command is read from the keyboard and carried out. If the input is a move, the program makes the move (and says "congratulations" if the game is won). Then, if the game isn't over, the program finds a move and makes it.

The Data Structure

The size of the board is contained in the constant *N*. If you change this constant, the whole program changes accordingly. *Board* contains the board position, *Player* contains code signifying the player who is next to move, and *Line* and *Value* contain the tables for incremental updating. *TotalLines* contains the number of empty lines left (the game is a draw when this number is 0). *Weight* is discussed under the *Add* procedure, and *AttackFactor* is discussed under the *FindMove* procedure later in this chapter.

Below is the source code containing constant, type and variable declarations for the Go-Moku program.

```
program Gomoku;
const
  N           = 19;                                    { Size of the board }
  Esc         = #27;
  CtrlC       = #3;
  Return      = #13;
  Space       = #32;
  AttackFactor =  4;                          { Importance of attack (1..16) }
  Weight      : array[0..6] of              { Value of having pieces in a row }
                integer = (0, 0, 4, 20, 100, 500, 0);
  NormalColor : integer = White;
  BorderColor : integer = Yellow;
  BoardColor  : integer = Cyan;
  HeadingColor : integer = Brown;
type
  BoardType   = (Empty, Cross, Nought);              { Contents of a square }
  ColorType   = Cross..Nought;                           { The two players }
  IndexType   = 1..N;                                  { Index to the board }
  NumberType  = 0..5;                              { Number of pieces in a line }
  LineType    = array[ColorType] of           { Value of square for each player }
                NumberType;
  ValueType   = array[ColorType] of integer;
  MaxString   = string[255];              { Used only as a procedure parameter }
```

```
var
  Board      : array[IndexType, IndexType] of BoardType;          { The board }
  Player     : ColorType;                    { The player whose move is next }
  TotalLines : integer;                     { The number of empty lines left }
  GameWon    : boolean;                  { Set if one of the players has won }
  FileRead   : boolean;                { Help file read? ... Help system ... }
                              { Number of pieces in each of all possible lines }
  Line       : array[0..3, IndexType, IndexType] of LineType;
                                    { Value of each square for each player }
  Value      : array[IndexType, IndexType] of ValueType;
  X, Y       : IndexType;                                    { Move coordinates }
  Command    : char;                                { Command from keyboard }
  AutoPlay   : boolean;                  { The program plays against itself }
```

Go-Moku Procedures

This section covers all the procedures in the Go-Moku program. The procedures *SetupScreen*, *PrintMove* and *ResetGame* do what their names imply: the first draws the initial screen display for a new game, *PrintMove* places a move on that display, and *ResetGame* returns all variables to their original values.

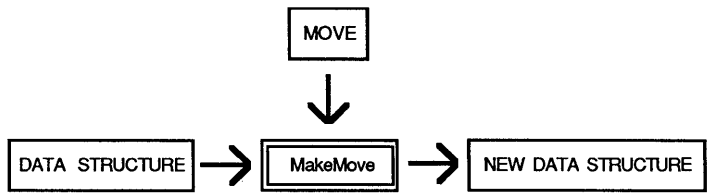


Figure 5-5. How the MakeMove Procedure Works

MakeMove

MakeMove is the procedure responsible for incremental updating. A typical game situation would be that *Player* has made the move *(x,y)*, so that horizontal lines in *Line* and *Value* are updated. *Line* is organized by the four possible directions (horizontal, vertical, right/left diagonal) and the starting point of each line. Horizontal lines go from left to right, so *Line[0,1,1]* contains the numbers for the line from *A1* to *E1*, and *Line[0,15,8]* the number for the line from *O8* to *S8*. So if the move is, for instance, an X at *O8*, we should add 1 to the number of X's in the lines starting at *K8* to *O8* (the numbers found at *Line[0,11,8,Cross]* to *Line[0,15,8,Cross]*). It takes the five lines one by one, calculates the starting point in *(x1,y1)*, and adds 1 to *Line[0,x1,x1,Player]*. Then it updates *Value* for each of the five grid intersections in the line (actually only the values for the empty grid intersections need to be updated because you won't be placing a piece on a grid intersection where a piece already exists). If the line starts at *O8*, we have to *update* the values for the squares from *O8* to *S8*, and the inner loop does that.

Here is the source code for the *MakeMove* procedure.

```
begin                                                          { MakeMove }
  LineCode := -1
  Opponent := OpponentColor(Player);
  GameWon  := false:
  { Each square of the board is part of 20 different lines.
    The procedure adds one to the number of pieces in each
    of these lines. then it updates the value for each of the 5
    squares  in each of the 20 lines.  Finally board is  updated,
    and the move is printed on the screen. }
  for  K := 0 to 4 do                    { Horizontal lines, from left to right }
  begin
    X1 := X - K;                                    { Calculate starting point }
    Y1 := Y;
    if  (1 <= X1) and (X1 <= N - 4) then               { Check starting point }
    begin
      Add(Line[0,  X1, Y1, Player]);                        { Add one to line }
      if GameWon and (LineCode < 0) then
        LineCode := 0;
      for  L := 0 to 4 do          { Update value for the 5 squares in the line }
        Update(Line[0, X1, Y1], Value[X1 + L, Y1]);
    end;
  end;                                                                  { for }
  for K := 0 to 4 do           { Diagonal lines, from lower left to upper right }
  begin
    X1 := X - K;
    Y1 := Y - K;
    if (1 <= X1) and (X1 <= N - 4) and
       (1 <= Y1) and (Y1 <= N - 4) then
    begin
      Add(Line[1, X1, Y1, Player]);
      if GameWon and (LineCode < 0) then
        LineCode := 1;
      for L := 0 to 4 do
        Update(Line[1, X1, Y1], Value[X1 + L, Y1 + L]);
    end;
  end;                                                                  { for }
  for K := 0 to 4 do                          { Vertical lines, from down to up }
  begin
    X1 := X;
    Y1 := Y - K;
    if (1 <= Y1) and (Y1 <= N - 4) then
    begin
      Add(Line[2, X1, Y1, Player]);
      if GameWon and (LineCode < 0) then
        LineCode := 2;
      for L := 0 to 4 do
        Update(Line[2, X1, Y1], Value[X1, Y1 + L]);
    end;
  end;                                                                  {for }
  for K := 0 to 4 do           { Diagonal lines, from lower right to upper left }
```

```
  begin
    X1 := X + K;
    Y1 := Y - K;
    if (5 <= X1) and (X1 <= N) and
       (1 <= Y1) and (Y1 <= N - 4) then
    begin
      Add(Line[3, X1, Y1, Player]);
      if GameWon  and (LineCode < 0) then
        LineCode := 3;
      for L := 0 to 4 do
        Update(Line[3, X1, Y1], Value[X1 - L, Y1 + L]);
    end;
  end;                                                    { for }
  Board[X, Y] := Player;                          { Place piece on board }
  if GameWon then
    BlinkWinner(Player, X, Y, LineCode)
   else
    PrintMove(Player, X, Y);                        { Print move on screen }
  Player := Opponent;                       { The opponent is next to move }
end;                                                      { MakeMove }
```

Add and Update

Add adds 1 to the number of pieces in a line, and updates *TotalLines* and *GameWon*. *Update* is a bit more complicated. The value that results from expanding a line to *N* pieces or blocking a potential opponent line of *N* pieces is found in *Weight[N]*. As mentioned before, there are two cases. First, if the *Opponent* has no pieces on the line, then *Valu[Player]* contains the value of expanding the line to *Lin[Player]* pieces (since we have just added one to *Lin[Player]*). We therefore have to update *Valu[Player]* from *Weight[Lin[Player]]* to *Weight[Lin[Player]+1]*. Second, if the *Opponent* has pieces anywhere along the five-position line, and it is the first piece that *Player* places on the five-position line, the line is no longer of value for the *Opponent*, and we update *Valu[Opponent]* from *Weight[Lin[Opponent]+1]* to 0. Once both players occupy positions in the same five-position line, neither can win with it—so the line is no longer of value to either of them.

We've just examined a sophisticated game strategy contained in only a few lines of code. The rest is easy. After handling the possible horizontal lines, *MakeMove* does exactly the same with the vertical and diagonal lines. Finally, it places the piece on the board, prints the move on the screen, and changes *Player* to *Opponent*. *Player* and *Opponent* change places each move (the X's and O's change with them), so that the program can give a hint to the current player or make its own move by using the same code. After all, both program and human *should* pick the best spot for their next piece. Why not use the same logic?

Next, there is a short function called *GameOver*. You might wonder why we don't use a variable which is updated in *MakeMove*, as with *GameWon* and *TotalLines*. The reason is that the same information would be stored twice. Since execution time is not essential here, there is no reason to store information that is already available; it would just give us an extra variable to remember to update. This way, it is updated automatically. Remember this trick—we are going to use it again later on.

FindMove

FindMove determines which potential move has the highest evaluation. Note how the evaluation is calculated, using the values for *Player* (expanding lines) and *Opponent* (blocking lines). The value for expanding carries a greater weight than the values for blocking; exactly how much higher is determined by *AttackFactor*. The rest of the *FindMove* procedure handles input and output.

Try experimenting with *AttackFactor*. A value of 4 weights attacks 25% higher than defenses. A value of −16 would make the program ignore its own attacks and concentrate completely on blocking. If you give it a high value, the program will concentrate on its own attacks. The program uses *AttackFactor* to weight the balance between attack (making lines) and defense (blocking lines). Isolating this factor in a constant makes it easier for the game designer to determine an optimal value for best play.

You could try different values here, and watch the effect on the program's playing ability. You can even convert *AttackFactor* into a variable and set it from the keyboard when you start each game.

Below is the source code for the *FindMove* procedure.

```
procedure FindMove(var X, Y : IndexType);
{ Finds a move X,Y for player, simply by
  picking the one with the highest value }
var
  Opponent  : ColorType;
  I, J      : IndexType;
  Max, Valu : integer;
begin                                                    { FindMove }
  Opponent := OpponentColor(Player);
  Max := -MaxInt;
            { If no square has a high value then pick the one in the middle }
      X := (N + 1) DIV 2;
      Y := (N + 1) DIV 2;
      if Board[X, Y] = Empty then Max := 4;
                { The evaluation for a square is simply the value of the
                  square for the player (attack points) plus the value for
                  the opponent (defense points). Attack is more important
                  than defense, since it is better to get 5-in-line
                  yourself than to block your opponent's moves.          }
  for  I  := 1 to N do                                { for all empty squares }
    for J := 1 to N do
      if Board[I, J] = Empty then
      begin                                                { Calculate evaluation }
        Valu := Value[I,  J,  Player] * (16 +  AttackFactor) DIV
                16 + Value[I, J, Opponent] + Random(4);
        if Valu > Max then                      { Pick move with highest value }
        begin
          X := I;
          Y := J;
          Max := Valu;
        end;
      end;                                                             { if }
  end;                                                          { FindMove }
```

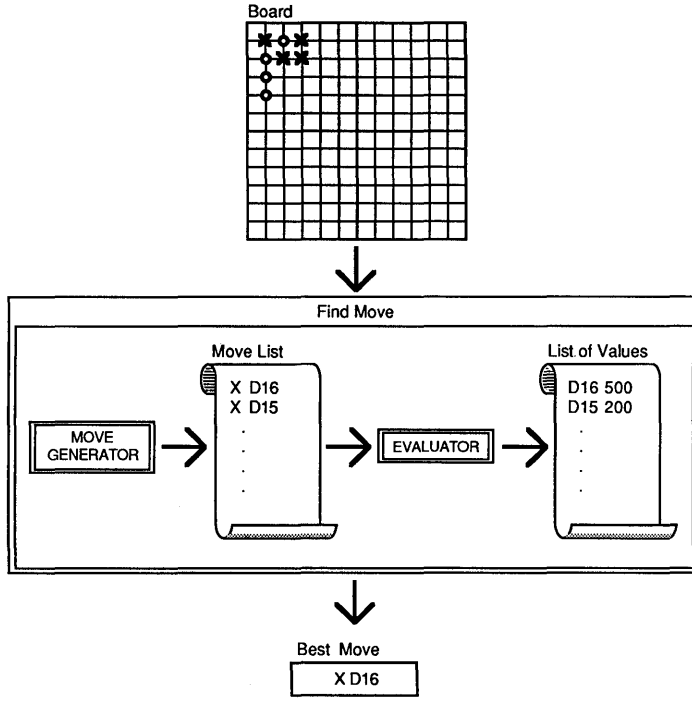


Figure 5–6. How The FindMove Procedure Works

By now you should have a good understanding of what makes Go-Moku tick. It's a good program to start with—short and uncomplicated. Yet it illustrates most of the principles all games must follow: an evaluation function, a data structure that represents the scope of play, and a stylistically adequate user interface so that playing the game itself is interesting. In addition, an on-line help module is included that retrieves and displays text from the ASCII file GO-MOKU.HLP. All the statements in GO-MOKU.PAS that support the help system are marked with the comment: ...Help system... or are in the GO-HELP.INC include file.

Chapter 6
CHESS PROGRAM DESIGN

This chapter takes you on a procedure-by-procedure tour through the Turbo Chess program. We'll explain how the program evaluates and generates moves, and give you ideas about how to alter the Turbo Chess source code or write chess programs of your own.

We begin by examining some of the principles used in developing a chess program—evaluating moves, searching, selecting and generating moves, and keeping track of time. The last part of the chapter discusses each Turbo Chess module in detail.

Turbo Chess can help you learn strategic game programming by example. It ranks as a "pretty good" program. You may find that other chess programs play more strongly or more quickly—but with Turbo Chess you have access to the source code and a guided tour of the techniques used. It's up to you to modify the program and make your version the best computer chess program in the galaxy!

A faster program (i.e., an assembler program) may play better chess because it can examine more potential moves in a given amount of time. ELO points are the scale used to measure playing strength of chess players. A difference of 100 ELO points between two players means that the better player should win two-thirds of the time. A factor of two in speed translates to about 60 ELO points in playing strength. A factor of ten (about the margin that hand-coded assembler programs have over Turbo Chess) means that an assembler program has a 200 ELO point edge.

The first requirement of any strategic game—and especially chess—is to create a data structure that represents each position and move. We will need a procedure to calculate the possible moves from a given position for a given piece, a procedure to play a move from a given position, and a number of other small procedures to handle I/O and other housekeeping details. A procedure that "understands" chess piece direction and legal moves, one that makes a legal move, and procedures that handle screen and disk I/O are pretty standard programming fare. The key to the Chess program—as it was with the Go-Moku program—is an evaluation function to decide *which* move is best in a given situation.

Evaluating Moves

Piece	*Worth In Pawns*
Pawn	1
Knight	3
Bishop	3
Rook	5
Queen	9

The evaluation function, which contains the program's "understanding" of the game, calculates the value of a given position and then determines the program's moves based on the values it computes. The results of its computations are displayed in the *value* box in the control panel area of the screen. By convention, a zero value means the human and computer players are even in score, while a positive value means that the program has an advantage. The most important factor in the evaluation function is *material*. Material is a list of the number and rank of chess pieces a player has at a given time. Pawns, knights, bishops, rooks and queens are roughly worth 1,3,3,5 and 9 points, respectively. The material balance is normally weighted higher than any of the other parts of the evaluation function. Many programs, including this one, *never* sacrifice material to get a positional advantage.

The *positional* part of the evaluation function decides the relative values of moves. Every programmer has his or her way of doing this; it is partly a matter of chess-playing style. Normally, you put in those things that you consider important when you play chess yourself.

Three important factors that most chess programs include in the positional part of the evaluation function include:

- *Mobility*—the number of squares each piece is able to attack.
- *Pawn structure*—advancing pawns, control of board center, doubled or isolated pawns, connected pawns and passed pawns.
- *King attack/safety*—pieces that can attack squares near the opponent's king or pawns around the king.

In addition to these three important factors, there are a few other factors you can put in your program: a bonus for castling, a bonus for exchanging pieces when ahead, placing rooks behind passed pawns, moving the king to the center in the endgame, etc. Note that the evaluation is symmetric: if the program gets a bonus for attacking the opponent, it also gets a penalty if the opponent attacks the program. Each factor is given a weight, and all the different values for a position are added to produce a final evaluation of the position. It can be a big problem to determine all the different weights. In many programs, the weights depend on the game phase (for example, king safety is important in the middle of the game but not in the endgame, and passed pawns are normally not important until late in the game).

The mobility factor might sound strange to you, since it is not normally considered important in chess. However, many of the standard maneuvers in chess (rooks on open or semi-open files, rooks on 7th rank, bishops on long diagonals, knights in the center, moving pawns forward to gain space) can actually be seen as ways to improve mobility.

The evaluation functions used in Turbo Chess are for the most part rather simple, at least compared with the playing strength of the best programs. Creating a good evaluation function is a very demanding task; making a uniformly good evaluation function is next to impossible. Yet it is with the evaluation function that the key to a better program lies.

Below is the source code for the *Attacks* function (part of the evaluation code):

```
function Attacks(AColor: ColorType; Square : SquareType) : boolean;
                                        {Calculates whether AColor attacks the square}
var
  i : IndexType;
begin
  Attacks := true;
  if PawnAttacks(AColor, Square) then {True if pawn attacks the passed in square}
    Exit;
                    {Other attacks:  Try all pieces, starting with the smallest}
  for i := OfficerNo[AColor] downto 0 do
    with PieceTab[AColor, i] do
      if IPiece <> Empty then              {Piece for the current board position}
        if PieceAttacks(IPiece, AColor, ISquare, Square) then
          Exit;
  Attacks := False:
end;                                                                  {Attacks}
```

The Essence of Chess: Looking Ahead

In the more simple Go-Moku program, all we really needed was the evaluation function: we simply picked the move that resulted in the position with the highest value and lived in the present. In chess, however, it is essential that our program look ahead and consider what is likely to happen in the next few moves. It does not help to win the opponent's queen if you are mated one move later. Very often you will choose combination moves in which you sacrifice a piece, only to get it back "with interest" a few moves later. To perform evaluations like this, it is necessary to look ahead. How do we make a program do that?

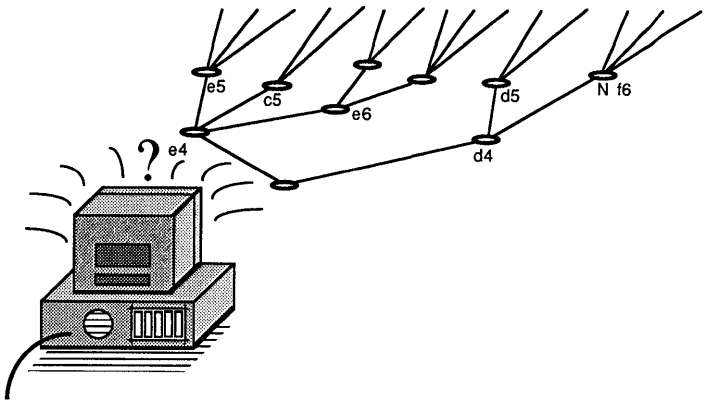


Figure 6-1. Searching for a Move

The Minimax Search

Let's start simply. On average, there are about forty possible legal moves for each chess position. After each of these forty moves, the opponent has about forty different moves to choose between. One approach is to try all the forty moves for the program, and for each of these moves try all the forty moves for the opponent. This is a one-move (or two-ply) lookahead. For each of its forty potential moves, the program looks for the one with the best evaluated value (remember that Turbo Chess evaluates factors such as position and material). Next, the program selects this best move and evaluates all of the opponent's potential forty moves. This time, however, it is looking for a condition in which the opponent gains very little (or even better: nothing or a negative value), even though it is the best available move. If the best move for the program results in an *even better* potential move for the opponent, the program selects its next best move, and evaluates the opponent's responses. Ideally, the program is searching for a move to make that will leave the opponent with the least satisfactory best move himself. This is called a *Minimax search*—it minimizes the opponent's advantage and maximizes the program's. We can also look more than two half-moves ahead, which would improve the game but take more time.

The algorithm below (written in Pascal-like pseudocode) should give you an idea of how the search is performed:

```
procedure Search;
begin
  if final depth then              {If we've gone as many plies ahead as we can, }
    evaluate position                                  {what is the evaluation? }
  else
  for move:=all possible moves do  {If not, move each piece programmatically in }
  begin                            {succession, evaluate the resulting position, }
    Perform (move);                    {and "take back" the move without telling }
    Search;                            {the opponent until we've hit on the best }
    TakeBack(move);                                          {move to make. }
  end;
end;
```

How Minimax Helps Turbo Chess Select a Move

Below is a sample Minimax tree search. Each possible and resulting chess position is represented by a *node* (the dots), and each *branch* in the tree represents a possible move. We measure the depth of the tree in *plies* (half-moves). The lowest nodes are *terminal nodes*.

In position 4, black will play *Nc6* or *d6*, and position 4 is thus worth 1 point for white. Position 7 is worth 0 points, since black will play *Bc4*. In position 3, white will therefore play *Nf3* instead of *Bc4*, and position 3 is thus worth 1 point for white. In the same way we find that positions 10, 15, 23 and 29 are worth 4,4,0 and -4 points. Thus, positions 2 and 22 are worth 1 and 0 points (since black selects the lowest value). White should therefore play *e4*, which is worth 1 point. The expected continuation is *e4, e5, Nf3, Nc6* or *e4, e5, Nf3, d6*, both leading to a position worth 1 point.

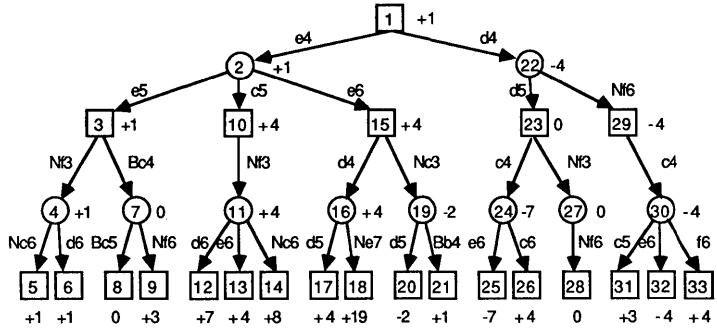


Figure 6-2. Sample Game Tree for the Opening Position

We can make a reasonable chess program by analyzing all possible moves to a depth of two to four plies (1-2 full moves). If there are pieces *en prise* (pieces that can be captured) at a terminal node, the program should take these pieces into consideration when deciding among moves.

The Minimax search looks for a potential move (for example, move 1 or move 2 in the diagram on page 52) that results in the lowest possible score for the opponent. In the diagram, move 1 produces a score of 6—but permits the opponent to make three possible moves, the best of which results in a possible score of 3. Move 2, on the other hand, only carries an evaluation of 3, but the best move the opponent could hope for carries a score of 1.

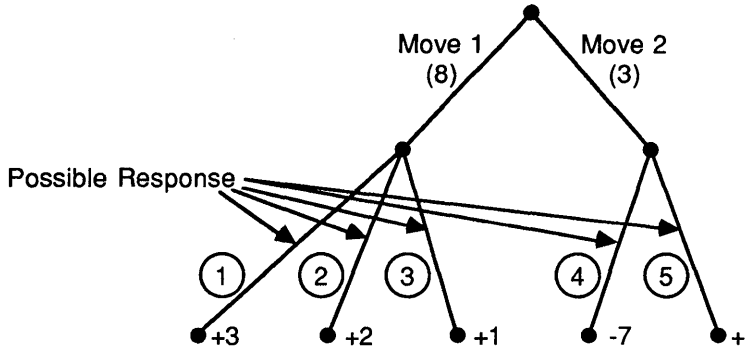


Figure 6-3. Sample Search Tree

When using Minimax, we assume that the opponent will use all his or her logic and intuition and select the best possible move available. S/he may not, however. In the case of move 2, we would be even further ahead. We can extend the search beyond one move to two or more full moves. Theoretically, the search can play out the entire game to decide between possible moves, but this would mean that the tree would rapidly grow beyond the capability of the computer or program to manage it. A four-ply search in chess would result in about one million terminal nodes.

Alpha-Beta Algorithm

The Minimax search is quite time consuming. Analyzing a chess position to a depth of four plies gives $40 \times 40 \times 40 \times 40 = 2{,}560{,}000$ terminal nodes—even when we ignore the capture search. Luckily there is a way to get around this, called the *Alpha-Beta* search.

The algorithm gets its name from the variables *Alpha* and *Beta* that compare evaluations. *Alpha* is initialized to $-\infty$ and *Beta* to $+\infty$. In actual practice, *Alpha* and *Beta* would be initialized to arbitrarily high and low numbers.

The Alpha-Beta algorithm prunes branches in a search tree as quickly as possible (evaluations permitting). It should give the same results as the more exhaustive Minimax search.

The object of the algorithm is to maximize the value of *Alpha* (the score of the program's own moves) and minimize the value of *Beta* (the score of the opponent's response). We've arbitrarily assigned

evaluation scores to the terminal nodes of the search (PVII through PXIV).

Most programs use a *quiescence* or *capture search* to analyze all captures to an unlimited depth. Thus, we first analyze all moves to a depth of 2-4 plies, then all captures until there are no more possible captures, and then evaluate the resulting terminal nodes with our evaluation function. This gives a good chess program. It is, in fact, how most chess programs work today.

Here's how the algorithm works:

- The program evaluates the PVII score as 2. This is greater than the level 2 *Alpha* of −infinity. *Alpha* becomes 2.
- The program wants to minimize *Beta* (the opponent). Because 2 < +infinity, level 1 *Beta* becomes 2. *Alpha* at level 0 becomes 2.
- PVIII scores 6. Level 2 *Alpha* then becomes 6 (6>2). . .but 6 fails to minimize the *Beta* at level 1. Move D fails (had there been succeeding moves, they too would have been ignored).
- PIX evalutes to 1. Level 2 *Alpha* still stands at 2 (because the prior node failed). PIX does not maximize the level 2 *Alpha*. Neither does PX, which evaluates to −3.
- PXI evaluates to −1 and PXII to 1. Neither of these will maximize the level 2 *Alpha* above 2. PXIII will, however—to 3. But the mission of the algorithm is also to minimize *Beta*. A 3 at level 1 *Beta* is greater than the 2 already there. The search terminates at this point, and the program selects move A.

The Alpha-Beta search can be speeded up still further if the terminal nodes are taken in descending order and checked out. That way, the best terminal chess position is checked first.

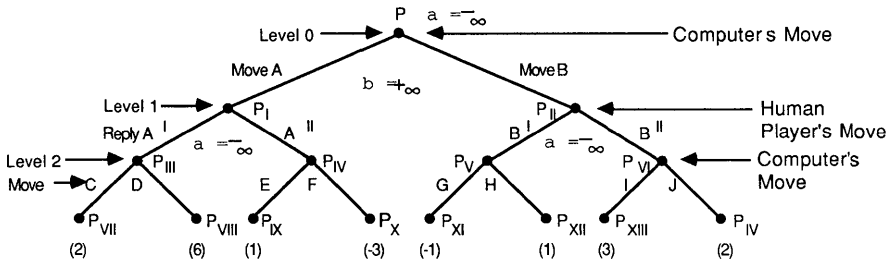


Figure 6-4. Sample Alpha-Beta Search

The Alpha-Beta algorithm works best if the best moves are always analyzed first. In that case, a one-ply search results in 40 terminal nodes as usual, but a two-ply search begins with an analysis of the first move (40 end nodes), and then may cut off the other 39 moves (39×1 terminal nodes). This totals only 79 terminal nodes instead of 40×40 = 1600 end nodes. A four-ply search ends up with 40×40 + 39×40 = 3160 terminal nodes instead of 2,560,000, a six-ply search 126,400 instead of 4,096,000,000 (that's over four trillion). In the Minimax search, each extra ply requires forty times more time.

In the Alpha-Beta search an extra ply takes approximately six times as long. The Alpha-Beta method can save an enormous amount of time.

Below is a short Pascal-like pseudocode function that shows how the Alpha-Beta algorithm is programmed. A four-ply search is performed with the call *Search(−MaxInt,MaxInt,4)*. This function is not in Turbo Chess; it is an example.

```
function Search(Alpha,Beta : integer; Ply : integer) : integer;
var
  BestMove : MoveType;                              { The best move          }
  MaxVal   : integer;                               { Value of best move     }
begin
  CaptureSearch := Ply <= 0;            { Shorthand: (ply <=0) evaluates T or F}
  BestMove := ZeroMove;
  MaxVal   := -MaxInt;                                { Initialize variables}
  if CaptureSearch then
  begin
    MaxVal := evaluation;                             { Calculate evaluation}
    if Alpha < MaxVal then
    begin
      Alpha := MaxVal;                  { Update alpha value                 }
      if MaxVal >= Beta then
      begin
        Search := MaxVal;                 { Check cut-off                    }
        Exit
      end;
    end;
  end;

  repeat
    if CaptureSearch then                { Generate next move                }
      Move := next capture
    else
      Move := next move;
    if no more moves then
    begin
      Search := MaxVal;                   { Check cut-off                    }
      Exit
    end;
    PerformMove                                          { Perform and}
    Value := -Search(-Beta,-Alpha,Ply-1);               { analyze the move}
    TakeBack(Move);                                     { Restore position}
    if MaxVal < Value then
    begin                                               { Update variables}
      BestMove := Move;
      MaxVal := Value;
    end;

    if Alpha < MaxVal then
    begin
      Alpha := MaxVal;                                  { Update alpha value}
```

```
      if MaxVal >= beta then
      begin
        Search := MaxVal;                          { Check cut-off }
        Exit
      end;
    end;
  until false;
end;
```

Iterative Search Before Alpha-Beta

The best move should be analyzed first if possible, although in practice it is enough that the first move analyzed be *one* of the best. One method of ordering best moves is the *iterative search*. As we've seen, a two-ply search takes 79 terminal nodes if the moves are ordered right, but can take up to 1600 terminal nodes if they are not. One way to order the moves would be to make a one-ply search before the two-ply search. In general, we could first make a one-ply search, then a two-ply search, then a three- ply search etc., until there is no more time. (Programs designed for large computers store most of the tree search from iteration to iteration, which requires several megabytes, but all we store is the best line.)

This method takes some extra time, but the time is saved again during the Alpha-Beta search because of better ordering. In addition, this method gives much better time control in tournament situations, because the best move from the previous search is always available. You can thus stop the Alpha-Beta search at any time and play the best available move when there is no more time left.

Tricks for Speedy Searches

We have described some basic search methods—iterative search, Minimax, Alpha-Beta, etc. There are several other small, but very important, tricks that speed up searches significantly.

An *Alpha-Beta Window* is the range of values outside which there will be a cut-off in the search. In the *Search* function, the Alpha-Beta Window is between the values *Alpha* and *Beta*, exclusive. Remember that the *Alpha* value must be maximized and the *Beta* value minimized when selecting a move. The variables *Alpha* and *Beta* hold the current best values found thus far in a given search. They are initialized to $-[\infty]$ and $+[\infty]$ (usually $-MaxInt$ and $+MaxInt$). A value higher than *Beta* causes a cut-off immediately. If the final value is lower than *Alpha*, it will cause a cut-off at a higher node in the tree.

If you study the source code, you will see that the *Search* function returns a value whether or not a cut-off has occurred. If the value is in the range between *Alpha* and *Beta*, no-cut off has occurred and

Current programs are very good at finding combination moves and playing aggressive chess, but they are generally weak at positional play. Positional play is the slow implementation of a chess strategy designed to give unquestioned dominance of the board and forestall attacks by the opponent. Capturing pieces is not as crucial, and can wait until much later in the course of a strong positional battle. This has not been a major problem until now, since most humans do not know how to play against a computer. Instead of playing calm positional chess, they try to wipe out the program with an attack. But if computers are to attain a level where they can play real matches in which players prepare before the games, they must gain a better understanding of the game through defined data structures and evaluation techniques.

the value returned can be used and processed. If the returned value is less than *Alpha* or greater than *Beta*, the value functions as a "flag" to tell the program that a cut off has occurred. If the returned value is higher or equal to *Beta*, this is called a *high cut-off*. If the value is lower or equal to *Alpha*, this is called a low cut-off. High cut-offs are normally very fast (often only one move has to be analyzed) while low cut-offs are somewhat faster than calculating the true value.

It is very important to analyze the best (or at least one of the best) moves first to speed up the Alpha-Beta search. We've seen how Turbo Chess tries to ensure this by generating the moves it will analyze in a particular order. It is essential to narrow the Alpha-Beta Window as much and as quickly as possible. We can also try to guess a window in advance. For example, once we have performed a three-ply search, it should be possible to guess quite accurately the evaluation for the four-ply search. Instead of initiating the search with Alpha and Beta equal to plus/minus *MaxInt*, we could start with a narrower window. If a move causes a high cut-off, we will have to analyze it again to get the exact value. If all moves cause low cut-offs, we will have to analyze all moves again to find a good move. The distance between the *Alpha* and *Beta* value is called the *width* of the window. The narrower the window, the faster the normal search will be, but when using this technique be aware that the risk of having to do recalculation is also larger.

Below is an example Pascal-like pseudocode function (not used in Turbo Chess) that shows a search using an Alpha-Beta Window. *Search* refers to a typical Alpha-Beta search function (not shown). A 4-ply search with window is performed with the call *WindowSearch (Alpha,Beta,4)*.

```
function WindowSearch(Alpha,Beta : integer;
                      ply        : integer) : integer;
var
  BestMove : MoveType;                          { The best move          }
  MaxVal : integer;                             { Value of best move     }
begin
  repeat
    BestMove := ZeroMove;
    MaxVal := -MaxInt;                           { Initialize variables   }
    OldAlpha := Alpha;

    repeat
      Move := next move;                           { Generate next move    }
      if no more moves then
      begin
        WindowSearch := MaxVal;
        Exit;
      end;
```

```
      Perform (move);                          { Perform and          }
      Value := -(Search(-Beta,-Alpha,Ply-1);   { analyze the move     }
      if Value >= beta then                    { Analyze move again   }
      Value := -(Search(-MaxInt,-Value,Ply-1);
      TakeBack(Move);                          { Restore position     }
      if MaxVal < Value then                   { Update variables     }
      begin
        BestMove := Move;
        MaxVal := Value;
      end;

      if Alpha < MaxVal then                   { Update alpha value   }
        Alpha := MaxVal;
    until false;

    if MaxVal <= OldAlpha then                 { Analyze all moves    }
    begin
      Alpha := -MaxInt;
      Beta := MaxVal;                          { again when low       }
    end;                                       { cut-off occurs       }
  until MaxVal > OldAlpha
end;
```

Principal Variation Search

An Alpha-Beta Window with zero width leads to the *Principal Variation Search*, which today is used by nearly all the best chess programs, including Turbo Chess. This search algorithm starts by analyzing the principal variation from the previous iteration. This line is searched with a full-width window. All other moves are searched with a zero-width window, since we assume that the principal variation is the best line.

When you search with a zero-width window, you always get a cut-off. When you get a low cut-off, you skip the move as usual. If one of the moves causes a high cut-off, this move must be analyzed again with a wider window to get the correct value. This happens quite often, but the time spent re-searching is less than the time that would have been spent with full-width searches. The algorithm speeds up many programs substantially, and keeps the performance of the Alpha-Beta search close to its theoretical minimum time (top performance with optimal sorting). Most of the speed increase probably comes from the *CaptureSearch* (Principal Variation Search is described in detail in *Relative Performance of the Alpha-Beta Search* by T.A. Marsland, ICCA paper November 1982).

Ken Thompson of Bell Laboratories believes that a move causing a high cut-off on the first ply should not be analyzed again; all you need to know is which move is the best, not the exact value of the move. Instead, you can continue the search with the same Alpha-Beta window. If another move *also* causes high cut-off, however, you will have to analyze both moves again to determine which is best. Most commercial chess programmers do not use the practice of selecting a

move without knowing its exact valuation (advocated by Thompson) because they want to be able to print out the evaluation and the expected continuation (the main line or principal variation) of the game.

Sometimes Second Best is Good Enough: The Tolerance Search

Turbo Chess uses an algorithm called *Tolerance Search* or the *Negative Alpha-Beta Window*. The program does not always have to play the best move; we are satisfied as long as it plays *one* of the best moves.

Consider a position from which two moves are equally good. There is a paradox here: the smaller the difference in value, the harder it is to determine which move is best. But when the difference is completely insignificant, the *calculation* of relative value takes the longest time. Instead of finding the absolute best move, the program picks a move that might (or might not) be a bit weaker than the theoretical absolute best move. How much weaker is determined by the size of the tolerance (in Turbo Chess, set to 8 positional points, or 1/32 of a pawn).

Below is an example Pascal-like pseudocode function that shows the Principal Variation Search with Tolerance Search. A four-ply search is performed with the call *PVS(4)*. This function is not in Turbo Chess; it is an example.

```
function PVS(Ply : integer) : integer;
var
  BestMove : MoveType;                          { The best  move     }
  MaxVal   : integer;                           { Value of best move }
begin
   if Ply <= 0 then
   begin                                            { CaptureSearch  }
     PVS := -Search(-MaxInt,MaxInt,Ply);
     Exit;
   end;
   BestMove := principal variation move;        { Start by analyzing }
   Perform(BestMove);                           { principal variation }
   MaxVal := PVS(Ply-1);
   if first ply then                                { Add tolerance  }
     MaxVal := MaxVal + Tolerance;                   { at first ply   }
   TakeBack(BestMove);
   repeat
     Move := next move;                         { Generate next move }
     if no more moves then                      { Exit when no more  }
```

```
    begin
      PVS := MaxVal;
      Exit;
    end;

    Perform(Move);                                    { Analyze the move      }
    Value:= -Search(-MaxVal-1,-MaxVal,Ply-1);
    if Value > MaxVal then                           { Analyze move again     }
      Value := -Search(-MaxInt,-Value,Ply-1);
    TakeBack(Move);                                   { Restore position      }
    if MaxVal < Value then                            { Update variables      }
    begin
      BestMove := Move;
      MaxVal := Value;
    end;
  until false;
end;
```

Using Selection

When you play chess yourself, you can't consider *all* possible moves. Chess players normally consider no more than three to six different first moves, and although they do look ahead, they seldom look at more than 30 to 40 different positions (and even the best seldom look more than three to four moves ahead). Strangely enough, this goes for both weak and strong chess players.

If you look at a normal chess position, there are many moves you would never consider making. First, about 20% of the possible moves are stupid sacrifices. Many other moves result in no gain or loss and have a net value of zero to the player making them. The theory goes that if we could avoid analyzing all these bad moves we could save much time. Instead of analyzing all forty moves at each node, we could double the search depth if we could restrict ourselves to analyzing only ten moves at each node. This strategy is called *Shannon-B*.

Turbo Chess, however, does not use Shannon-B. From practical experience, our chess programmers found that move selectors had to be very complex to discern good and useful moves from legal but non-useful moves. As selectors become more complex, programs get slower. Current state-of-the-art technique is to use an evaluation method called brute force and try *every* move, good or bad, useful or useless. This results in a much simpler program and opens the way for other types of time-saving tricks.

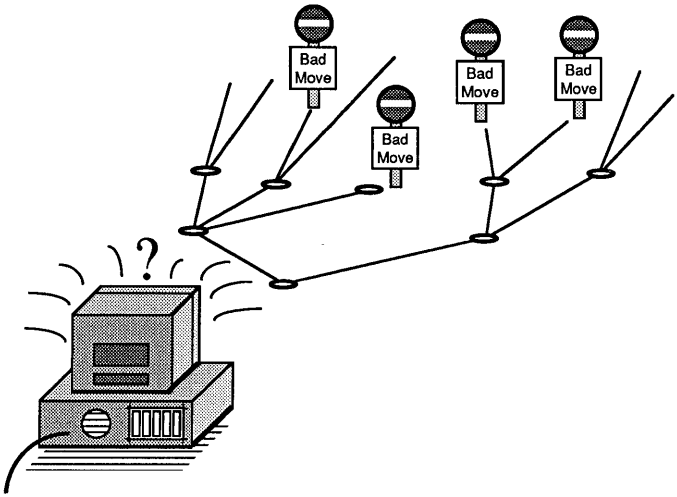


Figure 6-5. Searching with Selection

In tournament games, the best programs for large computers typically do an eight-ply search, while the best microcomputer programs do five to six plies (the Turbo Chess program does four to five plies).

While the theory of the move selector remains valid, a true move selector may have to wait for the advent of artificial intelligence.

You may wish to add some selector-like variations to Turbo Chess, however. One suggestion is to search an extra ply each time one of the players checks. A five-ply search can find a mate in five moves if all the attacking moves are checks.

Horizon Effect

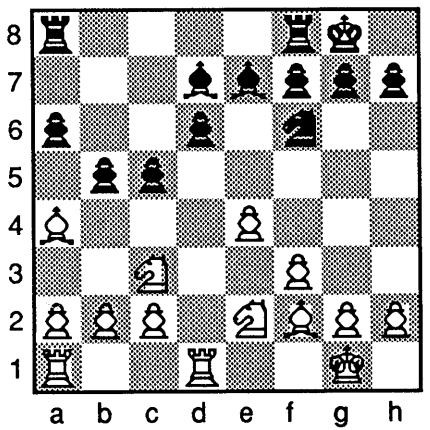


Figure 6-6. The White Bishop is Captured

Consider the position above: it is white's turn to move. It appears that unless the white bishop on *a4* is moved, it will be captured. With a three-ply search, the program will see that moving the bishop to *b3* attracts the black pawn to *c4* and *Bb3,c4* loses the bishop. Instead of giving it up, the program invents a foolish plan to save it by playing (*e5,PxP,Nd5*) white pawn to e5, black pawn on *d6* takes white pawn, white knight (on *c3*) to *d5*. This plan seems acceptable to the program; it sees that if *PxB* then *NxB*, or if *NxN* then *RxN, PxB, RxB*. The program believes it has saved the bishop, because the loss has been pushed beyond the three-ply search "horizon" (the program sees everything in the first three plies but nothing beyond three plies). In such a position, the program needs a six-ply search to come up with a reasonable plan—for instance, *Bxc5, bxa4, e5, Nb5, BxP*. This *horizon effect* is a very serious problem that pops up when working with multiple move strategy. There is also a positional horizon effect where no pieces are traded, in which a good positional move is completely ignored because it is pushed "beyond the horizon."

Program Design: a Closer Look

We are now going to look more deeply into some of the algorithms introduced in the previous sections. We are also going to look at some of the special tricks we can use to speed up our program.

Generating Moves

Most chess programs spend their time generating moves and handling the housekeeping chores that those moves create (evaluation, changing the content of array variables, etc.). Turbo Chess spends about 75% of its time this way. Thus, it's very important to generate moves quickly—that is, determine which pieces attack which other pieces. Some programs maintain tables with this information, which are then updated incrementally. However, it is just as fast to generate the moves directly from a given position. Although tables don't change very much when you move a single piece, it is still rather time consuming to update and restore them.

The chess *move generator* generates moves one at a time in an order that—logically—should result in finding the best moves first. At least it "plays the odds" in this regard. Remember that the evaluator uses an Alpha-Beta search technique to select the most advantageous chess move, and Alpha-Beta is considerably speeded by taking the potential moves in the best order we can provide for it. This list is an organized method designed to maximize the likelihood that a move we evaluate early will be the best move and thus halt the Alpha-Beta search quickly.

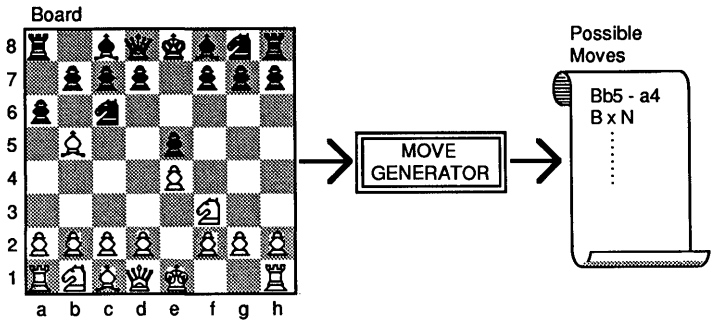


Figure 6-7. The Move Generator

Turbo Chess generates its moves in the order given below.

- *Best move from previous iteration.* The last time Turbo Chess made a move, the program projected the best moves that it and the opponent might make. This is the best place to start looking for the following best move, because we assume our opponent went through the same analysis as the program.
- *Capture of last moved piece.* About 20% of all moves are simple sacrifices. If this is the case with the opponent's last move, we take his piece and generate no further moves.
- *Killer Moves.* All moves that fail to cover a recognized threat are bad. If the program is threatened, it will ignore moves that fail to cover a threat. Once the threat is covered, moves may be analyzed further.
- *Other captures.* Captures of pieces other than the last moved piece.
- *Pawn promotions.* Moving a pawn to the rearmost rank of the opponent and trading it for a queen or other piece.
- *Castling.* Exchanging positions of rook and king.
- *Normal moves.* All moves other than the others listed here.
- *En passant captures.*

As mentioned earlier, we use the iterative search method. We start by analyzing the *best line from the previous iteration*. Second, we try to *capture the last piece moved by the opponent*. Since about 20% of

the possible moves from a chess position are sacrifices, this very simple rule cuts off nearly 20% of the moves. In the remaining 80% of the moves, the moving piece can often not be captured.

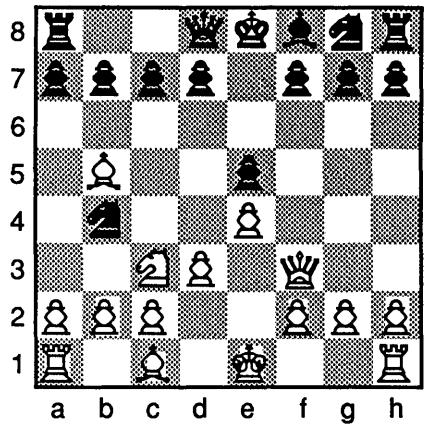


Figure 6-8. Black Threatens Nc2

Thirdly, we try something called *killer moves*. Consider a situation in which the opponent threatens our king and the rook on the next move with *Nc2* (Figure 6-8). The program must cover the threat; all moves that don't cover the threat are bad. When we analyze the first move for the program (*a3*), we find that it does *not* cover the threat; instead it gives the opponent an opportunity to make a very good move (*Nc2*). Then, when we analyze the program's second move (*a4*), it makes sense to see whether the opponent can still make the first very good move. If s/he can, then there is no reason to analyze the move further. If s/he can't, the move apparently covers the threat, and is worth analyzing further.

A move that is good from one position is very often also good from similar positions. When a move is found to be good in a position, either because it is the best move from the position or because it causes a cut-off, it is stored as a *killer move*. We store two killer moves at each ply-depth of the tree. If we stored only one, threats would be "forgotten" each time there was a move that covered the threat. Moves found to be good at one depth are tried again on the same depth, but not at different depths.

After killer moves, all other moves are generated. It is usually a good idea to start with *captures*; they must be analyzed during the capture

search anyway. We'll try to capture the largest pieces first. *Castlings* are good moves when they are possible. *Normal moves* are the majority of moves (not captures, killers, castling, etc.). They are generated starting with pawn moves and ending with king moves. Some programs generate the pawn moves last, some generate moves from the center and out. The order here is not very important, since at this stage you will probably have to generate all the moves anyway. Finally, *en passant captures* are generated.

Chess Evaluation Function

In the early days when chess programs were written exclusively for large computers, people designed large and complicated evaluation functions. Today, very few programmers use more than twenty different features. If evaluation functions get too large, they are too difficult to understand. The programmer must understand exactly what the evaluation function does, and that is impossible if the evaluation function becomes too large. An evaluation function should be as general as possible, so that it encompasses every possible position or situation.

The main provisions in the evaluation function in the Turbo Chess program are *piece development* (mobility) and *pawn structure* (their ability to guard and capture). For traditional reasons, a pawn is worth 256 positional points. A bonus of 64 points is worth 1/4 pawn.

Instead of weighting all squares equally when calculating mobility, each player and each square are assigned an *attack value*, which is the relative importance resulting from an attack on that square by that player. The attack value of a square depends on its closeness to the center of the board: *2 − 3/4 * (distance to center)* and the rank (in early middlegame *0.75, 1.5, 3, 3* points for 5th, 6th, 7th, 8th rank). In addition, the 8 squares around the opponent's king are assigned extra attack value (in the early middlegame *3* points).

An attack on the opponent is highest in attack value in the early middlegame, and decreases to zero in the endgame. The attack value could depend on the opening (for example, giving queen side attacks is more valuable than king side attacks for black in the Sicilian Defense). You could also let the program find the opponent's weak spots (weak pawns), and then give extra value for attacking those spots. This a good way to improve positional play.

The *positional value* of a *queen, rook, bishop* or *knight* is *the sum of the attack values for the squares which the piece attacks*. For *queens* however, the sum is multiplied by *1/4* (since the queen otherwise becomes much too aggressive). The positional value for *knights* is the sum multiplied by *1/2*, and *3 * (distance to center)* is subtracted from the value. The positional value of a *bishop* is normally a bit higher than that for a knight. Indirect attacks (a rook attacking through

another rook) are counted as normal attacks, attacks through blocking pieces (a bishop attacking through a knight) are counted at half value.

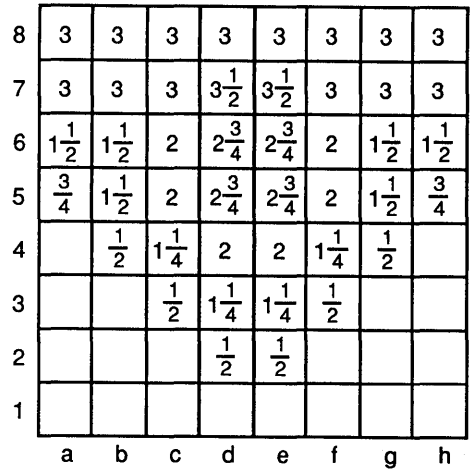


Figure 6-9. Attack Values For White in the Starting Position

The positional value of a pawn is equal to (*(PawnRank [Rank] + (Rank + 1) * PawnFileFactor[File]). The PawnRank* values are *0, 0, 2, 4, 8, 30* for 2nd to 7th rank (*plus 0, 10, 20, 40, 60, 70* if the pawn is a *passed pawn*). The *PawnFileFactor*s are *0, 0, 2, 5, 6, 2, 0, 0* for the a to h-file, respectively. Thus, a white pawn on *e2* is worth 0+(2+1)*6 = 18 points, while a pawn on *e4* is worth 2+(4+1)*6 = 32 points.

The value of the *pawn structure* is calculated as the sum of the individual pawn values. An *isolated pawn* gets a penalty of *20* points, a *double pawn* a penalty of 8 points, and an *isolated double pawn* a total penalty of 68 points. The program should also keep its pawns next to each other to build *pawn chains*. A pawn is given a bonus of 6 points if another pawn is on one of its sides; otherwise, it is given a bonus of 3 points if it is being covered by a pawn. A pawn is also given a bonus of 3 points for each pawn it covers. A wise chess master once said that every pawn move weakens your game. Because of this, the program is given a penalty of 3 points for moving a pawn. Moving a pawn from *e2* to *e4* would decrease the pawn structure value of the pawn from 6 to −3 points (6 for being next to another pawn, −3 for having moved).

Bonuses and Penalties in Positional Play

Turbo Chess gives bonuses and deducts penalties to assess the value of a given positional arrangement.

Positional Values

Pawns—Worth 256 Positional Points
64 point bonus = 1/4 pawn

Position in rank and file:

RANK	VAL.	+ PASSED PAWN?	+ PAWN ON ONE SIDE?	+ IS COVERED BY ANOTHER PAWN?	+ DOES COVER ANOTHER?	+ IF IN FILES A,B,G,H	+ IF IN FILES C,F	+ IF IN FILE D	+ IF IN FILE E
2nd	0	0	↑	↑	↑	0	2	5	6
3rd	0	10	↑	↑	↑	0	2	5	6
4th	2	20	6	3	3	0	2	5	6
5th	4	40	6	3	3	0	2	5	6
6th	8	60	↓	↓	↓	0	2	5	6
7th	30	70	↓	↓	↓	0	2	5	6

Penalties:
Isolated pawn = −20
Double pawn = −8
Isolated double = −68
Moving any pawn = −3

Pawn Structure = Sum of all pawn values

Queen, Rooks, Bishops, Knights

Attack value of a square: 2 − (.75 *(distance to center)) + rank factor rank factors: 5th – 0.75
6th – 1.5
7th – 3.0
8th – 3.0

The eight squares around the opponent's king rate an extra 3 points if they can be attacked.

Positional value of Q,R,B,N = Sum of attack values for all squares the piece can attack, tempered by the following factors:

Queens: attack value × .25
Knights: attack value × .5 − (3 × distance to center)

Rules for the Evaluation Function: "Evaluation Spices"

Every evaluation function must be tested thoroughly to see how it handles various game conditions. There are always a few special situations that a program will not handle properly. These can be adequately dealt with using what we call *evaluation spices*. An evaluation spice is a special rule for handling special situations. These rules should be *as narrow as possible* (in contrast to the general rules) to make sure a rule does not affect situations other than those in the special group. A common problem results when a rule inserted to solve one situation causes completely unforeseen changes in other situations which the program handled well before the rule was inserted.

Here are some evaluation spices that Turbo Chess uses.

- Normally, we want Turbo Chess to *castle short* (to the king's side) early in the game. For this, we give a bonus of 32 points. We also want to give a small extra bonus for *long* castling (to the queen's side—4 points). Quite often however, because it occurs so early in the game, the castle move is found in the opening library.
- When the program is ahead, it should *exchange pieces* (but not pawns) in order to reach a clearly won endgame; when the program is behind, it should *not* exchange pieces. There is a 32 point bonus for exchanging pieces when ahead.
- In the *endgame*, the program should *advance the king* to the center of the board. Kings are usually given a positional value of zero, but in the endgame they are given a penalty of (2 * *(distance to center)*).
- Placing a *rook behind a passed pawn* (either your pawn or an opponent's pawn) gives a 16 point bonus.
- Although the program is fond of *placing a bishop on d3 or e3* (since it attacks a lot of squares from there), this is a bad idea if it blocks a center pawn on *d2* or *e2*. This problem can be solved simply by giving a 20 point penalty for blocking a center pawn on *d2* or *e2* with a bishop on *d3* or *e3* (and the same for black). This problem is so common among chess programs that nearly all programs contain special programming to handle it.

Below is the source code for an evaluation spice that handles castling:

```
function CastlingMovGen : boolean;                          { Castling moves }
var
  CastDir : CastDirType;                                { Direction of castling }
begin
  CastlingMovGen := true:
  with MovTab[Depth] do
  begin
    Spe := true;
    MovPiece := King;
    Content := Empty;
    for CastDir := Short downto Long do      {Try castling on king-s side first }
```

```
      with CastMove[Player. CastDir] do
      begin
        NewX := CastNew;
        Old := CastOld;
        if KillMovGen(MovTab[Depth]( then    { Check the legality of the move }
          if LoopBody then Exit;             { Exits function returning true }
      end;:                                                          { with }
  end;:                                                              { with }
  CastlingMovGen := False;           { Which means that castling is a good move }
end;:                                                        { CastlingMovGen }
```

Sometimes the opponent does not resign when the program has a winning position. In such cases, the program must be able to go ahead and mate the opponent. The program should be able to perform the set of *basic mates* (*king and queen* against king, *king and rook* against king, and *king and two bishops* against king). King, bishop and knight against king require both great search depth and some special programming not in the Turbo Chess program. The usual rules about piece development don't work for mating, because in a mate situation we're more concerned with restricting the opponent's possible moves than with taking pieces, dominating board center, maintaining pawn structure and the other strategies designed to get us to the endgame. A special *mating evaluation* function is included in the program, used only to mate the opponent. It centers around only three factors (apart from material), listed here in order of importance:

- *Opponent's king* should be as far *away from the center* as possible (the object is to press him out to the corner).
- The *program's king* should be *close to the opponent's king*.
- If the program has *passed pawns*, the program tries to advance them.

The program must also know something about *draw*. It should value a *stalemate* or a *repetition* at zero. In addition, a line causing an *immediate repetition* of a position—which also occurred four plies above in the tree—is also evaluated to zero. This allows the program to find some forced repetitions very quickly. The *second repetition* of a position is given a bonus/penalty equal to one half the current evaluation of the position. Thus, if the program is four pawns ahead, a second repetition gives a penalty of two pawns. There is also a bonus/ penalty of one quarter of the evaluation if *10 full moves* have been made *without capturing a piece or moving a pawn*. After *48 full moves*, the bonus/penalty increases to 3/4 of the evaluation.

Calculation Method

Calculating the entire evaluation function for each new position can take a great deal of time; this was how it was done a few years ago, and still is done in some programs. One way to speed things up is to use incremental updating for most of the calculations. We've seen how incremental updating works in the Go-Moku program. When you move a pawn under incremental updating, you need only recalculate the evaluation for that pawn and any pieces the pawn was blocking or now blocks.

Another method for increasing speed—used by the *Cray Blitz* program—stores parts of the evaluation in *hash tables*. A *hash table* is a special table that allows you to use an index to look up values faster than in a binary tree. When, for example, the value of a specific pawn structure has been calculated, the value is stored in a hash table. If the same pawn structure is found in a different position, the value can simply be looked up in the table. Unfortunately, hash tables often take up a lot of RAM.

About half the microcomputer-based chess programs—including Turbo Chess—are based on tables, while the other half are based on fast incremental evaluations. One advantage to tables is that you can use a very sophisticated and complicated evaluation function, without worrying about how much time it takes. For details about how the tables are derived from the evaluation function, you can study procedure CalcPVTable in the EVALU.CH module on your distribution diskette.

The fastest method is to use *Piece Value* tables. This method was first used in the early 1980's. A Piece Value table gives the positional value for a given piece and player for a given square. For instance, the value of the move *Nf3−e5* is simply the value of a knight on *e5*, minus the value of a knight on *f3*. The tables are calculated at the beginning of the search using the evaluation function described earlier.

Piece Value tables are derived as a first-order approximation from the evaluation function. For instance, when the program calculates the values for rooks, it tries to place a rook on each of the 64 squares, and calculates the piece development value for the square; these values are then listed in the table. For instance, the piece development of a white rook is zero on the starting square, *h1*, but somewhat higher on *h3*.

Advantages/ Disadvantages to Piece Value Tables

The only real problem with Piece Value tables is that they are not changed during a search, and can therefore be unreliable on especially deep plies that are far away from the original (on-the-board-now) position. For instance, the program might decide that *h2−h4* followed by *Rh1−h3* is a good idea in the starting position, since it has calculated that the rook is well developed on *h3*.

If you invoke the *Value* command, the program displays the Piece Value tables on the screen (but divided by 2.56, so that a pawn is worth 100 points instead of 256 points). Figure 6-10 shows the Piece Value tables for white in the starting position. As you can see, the best move for white is *Nf3−e5* with a value of $3-(-5) = 8$ points.

White King

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

White Queen

2	3	4	3	4	3	3	2
2	3	4	4	4	4	3	2
3	4	4	4	4	4	4	3
3	3	4	4	4	4	3	3
2	3	3	4	4	3	3	2
2	2	2	3	3	2	2	2
2	2	2	2	2	2	2	2
0	0	0	0	0	0	0	0

White Rook

9	9	11	10	11	9	9	9
4	6	7	9	9	7	6	4
9	10	10	11	11	10	10	9
8	8	8	9	9	8	8	8
6	6	5	6	6	5	6	6
4	5	5	5	5	5	5	4
3	4	4	6	6	4	4	3
0	0	0	0	0	0	0	0

White Bishop

2	3	4	4	4	4	3	2
4	7	7	7	7	7	7	4
3	5	6	6	6	6	5	3
3	5	7	7	7	7	5	3
4	5	6	8	8	6	5	4
4	5	5	-2	-2	5	5	4
5	5	5	3	3	5	5	5
0	0	0	0	0	0	0	0

White Knight

-2	2	7	9	9	7	2	-2
1	4	12	13	13	12	4	1
5	11	18	19	19	18	11	5
3	10	14	14	14	14	10	3
0	5	8	9	9	8	5	0
-3	1	3	4	4	3	1	-3
-5	-3	-1	0	0	-1	-3	-5
-7	-5	-4	-2	-2	-4	-5	-7

White Pawn

7	7	13	23	26	13	7	7
-2	-2	4	12	15	4	-2	-2
-3	-3	2	9	11	2	-3	-3
-4	-4	0	6	8	0	-4	-4
-4	-4	0	4	6	0	-4	-4
-1	-1	1	5	6	1	-1	-1

Figure 6-10. Piece Value Tables For White in the Starting Position

Keeping Track of Time

Human chess players do not spend an equal amount of time on each move. Some simple moves are played immediately, while others are difficult and important and require more time. The time control method used by Turbo Chess is fairly simple. Before each move, the program calculates a desired response time; the actual time spent by the program seldom varies by more than 50% from the desired time.

Human players spend most time on the middlegame, and so does the program. In tournament games, it typically spends about six minutes per move in the early middlegame, and three minutes per move later in the game. This seems to give more than three minutes on average, but the first moves are usually played from the opening library, and quite often the program saves time or responds immediately because it has analyzed during the opponent's time of reflection. The program never exceeds the tournament or user-specified time limit even though it may not match its desired time, unless it does not have time to finish a 1-ply search.

Program Structure: The What, not the How.

Now let's turn to the source code itself. Please refer frequently to the CHESS code while you read the following sections. Study the program with the *RUF method*: read it, understand it, forget it. Read the comments first, followed by the code. Then make sure you understand how each procedure works. Then forget everything specific about that procedure *except* what it *does*. When you read the rest of the program, you have to know *what* the procedure does, not *how* it does it. Luckily, most of the procedures are not relevant to chess strategy—they cover user interface, housekeeping procedures and procedures that take care of I/O details. These non-game procedures aren't covered here.

Turbo Chess consists of one main file (CHESS.PAS) and nine *include* files. The include files are placed in the CHESS.PAS program by the compiler in the following order: TIMELIB, BOARD, MOVGEN, DISPLAY, INPUT, SMALL, EVALU, SEARCH, and TALK. All include files have the extension .CH.

CHESS.PAS

The CHESS.PAS file contains a few global constants, types and variables such as the opening library data structure. It also makes sure the include files are included in the correct order. It functions more or less as a table of contents for the complete program.

The following sections describe each of the Turbo Chess include files.

TIMELIB.CH

The TIMELIB file contains small procedures for measuring time. They are used for tournament play or other situations when you wish to place time limits on the program. They are completely independent of the rest of the program, so they can be used in other programs of

your own making. The file contains a type, *ClockType*, and three procedures (*InitTime, StartTime* and *StopTime*), which take a variable of *ClockType* as a parameter.

The clock works like a stopwatch that can be started, stopped and reset. Like a stopwatch, you can halt the hands at any point and restart them again; only reset brings the hands back to zero. Reset a clock to zero with *InitTime*. To measure a time interval, call *StartTime* and *StopTime*. The field *TotalTime* (type *real*) in the clock contains the length of the time interval in seconds. If you call *StartTime* and *StopTime* again, the new time interval is added to the old. *Stoptime* does not really stop the clock, it only updates it. Thus, if you call *StartTime*, followed by several *StopTime* calls, each call updates the clock (so you can use *TotalTime*), but at the end only the last call actually counts, as if no other calls than the last had been made.

BOARD.CH

The BOARD file contains the data structures that represent the chess board and the procedures to manipulate these structures.

The array *Board* contains the contents of each square. With this array alone, we would have to search the whole array to find, for instance, the position of the white king. Since it is important to save time, there is also an array *PieceTab* that contains the locations of the different pieces. These two arrays should always be updated together to ensure that they contain the same information.

Turbo Chess only roughly simulates the shape of the chess pieces on the screen. High resolution graphics would make the board display more attractive, but causes headaches for programmers working on computers that use different graphic techniques. By simplifying our screen I/O, we can concentrate on the central issues of chess programs without doubling the size of the program with graphics procedures. If you would like to "dress up" Turbo Chess, try the Turbo Graphix Toolbox.

The squares on the chess board are not represented by two coordinates, or by a number from 0 to 63. Instead, we represent a square with a binary value in the range from 0 to 119 ($77 in hexadecimal), where bits 4–6 represents the rank and bits 0–2 represents the file. The file numbers (*a–h*) and rank numbers (1–8) are represented with values from 0 to 7. Moves are generated for a piece by adding a direction value to the square where the piece is located. When generating king moves, there are eight direction values, which are added to the original square. The program must constantly check to see if a piece has reached the edge of the board. This would not be easy using values from 0 to 63.

7	6	5	4	3	2	1	0
0	Rank (1-8)			0	File (a-h)		

8	$70	$71	$72	$73	$74	$75	$76	$77
7	$60	$61	$62	$63	$64	$65	$66	$67
6	$50	$51	$52	$53	$54	$55	$56	$57
5	$40	$41	$42	$43	$44	$45	$46	$47
4	$30	$31	$32	$33	$34	$35	$36	$37
3	$20	$21	$22	$23	$24	$25	$26	$27
2	$10	$11	$12	$13	$14	$15	$16	$17
1	0	1	2	3	4	5	6	7
	a	b	c	d	e	f	g	h

Figure 6-11. Representation of Squares

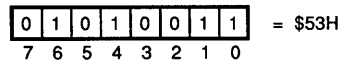
How Turbo Chess Represents the Board

A one-byte binary value in the range 0 to 119 (0H to 77H) represents each square. Within the byte, various bits act as indices to a square's rank and file.

The rank and file numbers are represented by binary values from 0 to 7. Seven is the largest number that can be represented by three bits.

RANK	FILE	BIT PATTERN
1	A	0 0 0
2	B	0 0 1
3	C	0 1 0
4	D	0 1 1
5	E	1 0 0
6	F	1 0 1
7	G	1 1 0
8	H	1 1 1

A square at position *d6* would have the following bit pattern and value:



Turbo Chess generates moves by adding a direction value (or a number of them, in the case of a piece with a complex pattern, such as a knight) to the value of the square on which the piece rests. The result is the value of the piece's new square.

A piecetype can be a king, queen, rook, bishop, knight, pawn, or it can be *empty*. The color can be either *White* or *Black*. For each square, the *Board* array contains the piecetype, the color and an index to *PieceTab*. If the piecetype is empty (empty square), the other values are undefined.

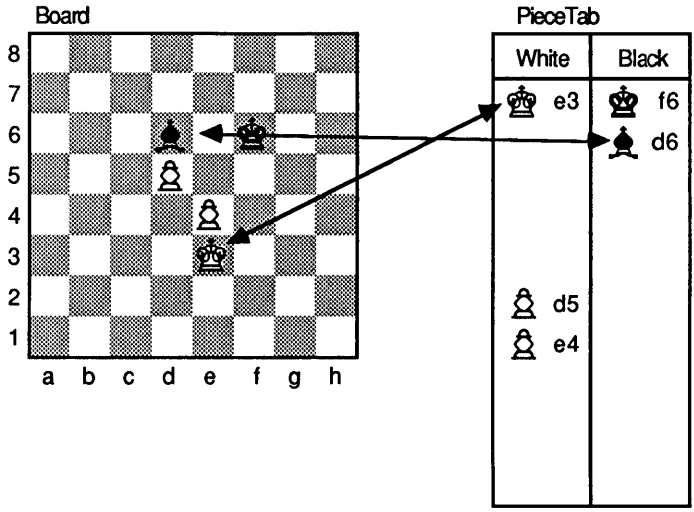


Figure 6-12. The Relation between Board and PieceTab

The program is organized in *modules*. A module is a set of data structures, along with procedures and/or functions that operate on the data structures. From outside a module, you can only access the data structures through the procedures and functions. In fact, when you use the procedures, you do not have to know exactly what the data structures look like.
Data structures in modules can be completely hidden from outside the module. The TIMELIB procedures are an example of a module. Since the data structures can only be changed via the associated procedures, no other part of the program can inadvertently affect the data structure. Modules are very helpful for structuring large programs. The best way to design modules is to first find out what operations you will do with the data structure, then design a data structure that fits your requirements. Try to keep the data structure as compact as possible, don't store the same information twice, and don't store data that can be calculated from other data. Finally, write the code for the procedures and functions.

For each chess piece, the *PieceTab* array is indexed with the color and an index from 0 to 15; the array contains the piecetype and the square it occupies. Pieces are sorted by value, with the king at index 0, and pawns at index 15. To speed things up, there are two indexes for each color, *OfficerNo* and *PawnNo*, which contain the indexes of the last officer and the last pawn. The officers (all pieces other than pawns) are placed from index *0* to *OfficerNo*, and the pawns are placed from index *OfficerNo + 1* to *PawnNo* (because of the way pawn promotion is implemented, the officers might not always be strictly sorted, and there might even be pawns placed among the officers).

We have just covered the twenty most important lines in our program. Now we must know which color is to move next. This color is contained in the variable *Player*, while *Opponent* contains the other player. To implement *en passant* capture, castling, three-fold repetition and 50-move rule, we must know more about a chess piece than just its board position. It would help to be able to look back at past moves—and if necessary count them.

Because Turbo Chess is a program, not a cutthroat tournament, a player should be able to backtrack through past moves, perhaps stopping five or ten moves back to play a game out differently.

The most flexible way to store the necessary information for castling, three-fold repetition or backtracking is to maintain a game history. The played game is stored in an array called *MovTab*, and the last played move is *MovTab[Depth]*. When you've asked for a hint, however, *MovTab[Depth]* instead contains the next move to be played. In the same way *Player* sometimes contains the player who has just moved. This might appear inconsistent, but it happens in only a few places in the program.

Moves are represented by a data structure called *MoveType*. A move consists of the *Old* (the square the piece is moving from) and *New* (the square the piece is moving to) squares. It is also useful to include the type of the moving piece (*MovPiece*) and the *content* of the new square (either empty or the type of the captured piece). All moves should thus be represented, including the three special moves (castling, *en passant* captures and pawn promotions). We discussed these in the section on evaluation procedure "spices." A variable *Spe* indicates whether the move is a special move. *Castlings*, which always involve moving both king and rook, are represented by the king's move, and *MovPiece* carries the value of king. Because kings can only move one square at a time *except* when castling, *New, Old,* and *Spe* tell us the move was a castle. *En passant captures* are represented with the new square of the moving pawn, and *MovPiece* carries the value of pawn. *Pawn promotions* are represented by the pawn move, and *MovPiece* contains the new piecetype (queen to knight).

Figure 6-13 shows an example for *MoveType*.

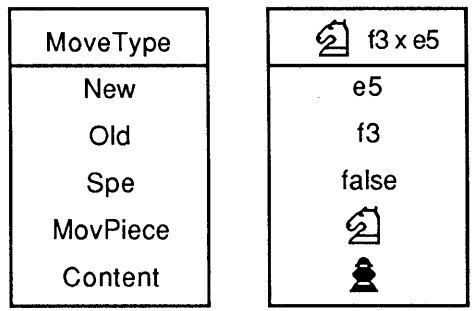


Figure 6-13. Sample Move Representation

There is a special move—the *ZeroMove*—that the move generator (*MovGen*) function produces to signal the program that it can produce no more legal moves to evaluate. *ZeroMove* acts as an "end of file" character.

The *EqMove* function checks whether two moves are identical.

The next procedure is *CalcPieceTab*. To set up a position, place it in *Board*. Then *CalcPieceTab* automatically calculates *PieceTab*, so that *Board* and *PieceTab* contain the same position.

Finally, the *Perform* procedure plays and takes back moves. The first parameter, *Move*, is the move. The second, *Reset*, is FALSE if the move is performed, and TRUE if the move is taken back. Performing and then immediately taking back a move results in exactly the same position. Inside *Perform* are four small subroutines, which change the board as follows:

- *MovePiece* moves a piece from one square to another
- *DeletePiece* removes a piece from the board
- *InsertPiece* inserts it again
- *ChangeType* is used for pawn promotion

With these procedures, it is easy to write the *Perform* procedure, although the three special moves (castling, en passant, and pawn promotion) require some special programming (as we've seen).

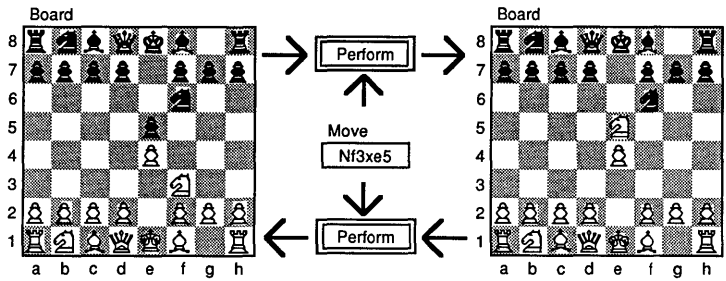


Figure 6-14. How the Perform Procedure Works

Having read and understood this code, you can now forget it all. The important information about Turbo Chess is that *Perform(Move,false)* performs a particular move, and *Perform(Move,true)* takes it back again.

MOVGEN.CH

The MOVGEN file contains procedures and functions to:

- generate moves
- test which squares are attacked by which pieces
- calculate information from the move table (castling status, 50-move rule and three-fold repetition).

The *Direction* arrays contain directions used in move generation. For example, *DirTab* contains the eight possible king and queen directions (left, right, up, down, and diagonals). The attack functions are as follows:

- *PieceAttacks* checks whether a square is attacked by a specific piece
- *PawnAttacks* checks whether a square is attacked by an opponent's pawn
- *Attacks* (which uses *PieceAttacks* and *PawnAttacks*) checks whether a square is attacked by the opponent

It is extremely important that attack functions be as fast as possible. The program spends about 20% of its time in procedure *PieceAttacks*, which is only twenty lines of code (to optimize the chess program, this would be a good section of the program to code in assembly language). The constant array *AttackTab* is a table which—when indexed with the difference between two squares—gives the pieces able to attack the one square from the other (if the area between the squares is empty). This trick makes the *PieceAttacks* procedure very fast (since in most cases it can be determined immediately that the square cannot be attacked by the piece). Note that bit manipulations are used to handle the moves from square to square.

Instead of searching the whole table every time *FiftyMoveCnt* is called, we could maintain and update the value in a variable, and could do similar things with *CalcCastling* and *Repetition* (in fact, most other chess programs work that way. A lot of programmers use hash tables for handling repetition of moves). However, this would make the data structure more complicated than necessary and require variables to be updated constantly. Instead, we stay true to the principle of not storing the same information twice.

Other procedures in MOVGEN are:

- *CalcCastling* checks if castling is illegal because a player has moved his or her king or rook
- *FiftyMoveCnt* counts the number of half moves since last capture or pawn move
- *Repetition* counts the number of times the current position has occurred before

The game is a draw if *FiftyMoveCnt* = *100* or *Repetition* = *3*. The 3 functions calculate the information by searching the move table (*MovTab*) each time they are called. This is time-consuming, but these functions are not called very often.

Move Generation Procedures

The function *KillMovGen* checks whether a specific move is possible from the current position. The move must be a legal move—i.e., there must be a position in which the piece can make the move. There are two nearly identical move generators in this program, since

it is difficult to use the same move generator for searches and for other purposes. The one described here is the one not used in the search (look for the other one in SEARCH). By convention, a move generator generates moves one at the time. The move generator is called *MovGen*, and it returns the move in a global variable called *Next*. When all moves have been generated, it returns the *ZeroMove*. The move generator is a very important part of the program, and it is a good idea to study it carefully if you want to thoroughly understand the program.

The easiest way to generate moves one by one is to cheat and generate them all at once, then store them in a buffer. Procedure *InitMovGen* does just that.

The *Generate* procedure places the next move in the buffer. The moves are generated starting with captures, followed by castlings, normal moves and *en passant* captures. *CapMovGen* generates captures of a specific piece (on the *New* square). *NonCapMovGen* generates non-capture moves for a specific piece (on the *Old* square).

DISPLAY.CH

For simplicity, the program uses a static screen picture, with the screen divided into different fields, each with a specific function. The *Message* field is used for messages: all messages are written in this field, and the field is never used for anything else. One could, of course, make a system where different windows pop up with information and questions, allowing the user to change the colors and the screen display.

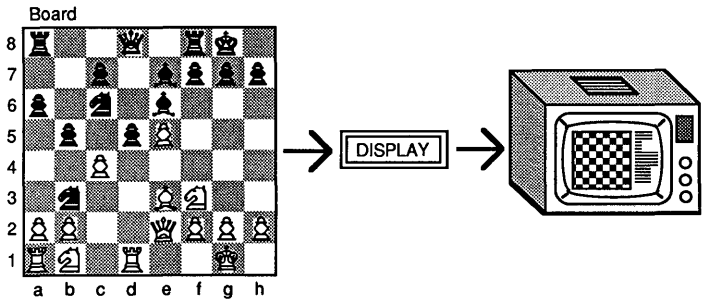


Figure 6-15. The DISPLAY Module

The screen display consists of two parts: a chess board and an information window. By altering the source code, you can define the colors and fields of the information window. Changing any of these colors changes the appearance of the screen dramatically.

Procedures that affect color are as follows:

- *NormalColor* is the color used in the information window
- *BorderColor* is the color of the border around the chess board
- *BoardColor* is the colors of the white and black squares (default green and red)
- *PieceColor* is the color of the white and black pieces (default white and black)

The screen and position constants (*NormalColor, BorderColor, BoardColor*, etc. for color; *HeadingPos, PlayerPos, LevelPos*, etc. for position) define the color and position of one of the fields in the information window. In the rest of the program, these fields are only accessed through the *GoToPos* procedure, which sets the color and places the cursor at the specified point in the field.

Next, there are a few small procedures, many of them only two or three statements. Small procedures are useful, not because they make the program smaller, but because they make it easier to change. Because of the **string** type used by Turbo Pascal, the procedures can be called with constant strings of different length. This is not possible in standard Pascal.

- *Message* prints a message.
- *Error* prints an error description.
- *Ask*, *ReadInput*, *ScanKeys* and *ReadCom* handle most of the input. The input is always placed in a global string variable called *Command*.
- *Ask* asks a question (such as "Move:").

Print Procedures

The print procedures are as follows:

- *ClearSquare* clears a square on the screen.
- *PrintPiece* prints the specified piece on the empty square. Piece pictures are quite easy to change.
- *SetupScreen* prints the screen display.
- *PrintBoard* prints the chess position found in *Board* (but does it without printing the whole board every time).
- *MoveStr* converts a chess move to a text string which can then be printed.

The following procedures display information in the different fields in the information window:

- *PrintTime, PrintMove* and *PrintNodes* print the chess clocks, the previous move and the number of analyzed nodes.
- *PrintBestMove* prints the main line (the best move found so far) and the evaluation.

This module also includes several procedures to clear and display:

- menus
- messages
- current player
- current level
- current mode (auto-play, multi-play or "normal")

INPUT.CH

This module contains many routines that communicate with the user and control keyboard input. A global variable, *CurMenu*, keeps track of which menu is calling the input routines. This is necessary because duplicate one-letter commands exist on several of the menus ("B" is back one move on the main menu, but changes the current color to black on the edit menu).

Another variable of scalar type, *CurOpt*, keeps track of which command option is currently selected (Back, Forward, Hint, etc.). This is needed so that commands can be checked in **case** statements (strings can't be used in **case** statements).

Both of these scalar variables allow Turbo Chess to use **case** statements for evaluating input and options.

INPUT routines are as follows:

- *ScanKeys* scans the keyboard, and if a key is pressed, it reads the input; otherwise, it returns immediately without doing anything. This procedure makes it possible to do calculations and print information, and at the same time read information from the keyboard.
- *ReadCom* prompts the user, sets *CurMenu* and calls *ReadInput*. Upon returning from *ReadInput*, *CurOpt* is set according to the values of *Command* and *CurMen*.
- *ReadInput* positions the cursor and calls *GetCommand* which then does the I/O.
- *GetCommand* takes input from the keyboard and updates the global variable *Command*. It utilizes a small army of procedures that enable the use of cursor control keys to make moves and edit pieces.
- *DisplayMove* displays information about the search. Normally it prints the search depth and the move currently being analyzed. In *SingleStep* mode, the whole search is displayed, one position at a time. If you experiment with the program you will probably

find the *Singlestep* mode very useful. It is also a very good way to learn what really goes on inside the chess program.

A large part of the INPUT module is made up of routines that are called to save the board when quitting the game or editing the board. One of these that may be of interest, *Hscroll*, reads a string in a one-line window and scrolls the input horizontally.

EVALU.CH

The EVALU file contains the evaluation module. The evaluation function and its parameters are described in detail on page 48. Most of the parameters (including the material value of each chess piece) are found in the top of the file, so it's easy to experiment with the evaluation function.

The module contains one major procedure and one major function:

- The *CalcPVTable* procedure calculates the *PVTable* (piece value) based on the chess position; *PVTable* is declared in the CHESS file. *CalcPVTable* also places the evaluation of the chess position in *RootValue*.
- *StateValu* function calculates the incremental static evaluation for a move using the *PVTable*.

Except for the following three features, the entire evaluation function is handled by the Piece Value table:

- Castling bonus
- Bonus for exchanging pieces when ahead
- Part of the pawn structure concerning isolated and doubled pawns; this part is calculated by incremental updating. The variable *PawnBit* represents the number of pawns of each color on each file. The function *PawnStrVal* uses *PawnBit* to calculate the value of the pawn structure for either color. *PawnBit* is updated by *StateValu* each time a pawn captures or is captured.

CalcPVTable puts values into the Piece Value table, a rather complex task. It is contained in about 300 lines of complicated code.

First the program calculates *Material* (material advantage for white), *TotalMaterial* (total material other than kings on the board), *PawnTotalMaterial* (total pawn material on the board), *MaterialLevel* (a measure of the material level of the game (early middlegame = 45 – 32, endgame = 0)) and *Mating* (set if the special mating evaluation should be used, in which case *LosingColor* contains the losing color). Then the program calculates *AttackVal*, which contains the attack value of the different squares for each player. *AttackVal* is used to calculate *PVControl*. For each color and square, *PVControl* contains the value resulting from rook or a bishop being on the square. This value is simply the sum of the attack values of the squares that a rook

or a bishop can control from the square, as described in the Program Design section. *PVControl* is used to calculate the *PVTable*; for rooks and bishops, the values are simply copied from *AttackVal* to the Piece Value table, and the values for the other pieces are calculated.

The next part of the code concerns pawn structure. First, *PawnBit* is initialized to the number of pawns of each color on each file. Then, the parts of the pawn structure handled by the Piece Value table are calculated—pawn chains, passed pawns, etc. There is also a penalty for blocking a center pawn with a bishop. Finally, the total evaluation for the position—its *RootValue*—is calculated.

The *StateValu* function apportions the positional bonus points among pieces. It is relatively simple because the evaluation function has already been calculated and placed in the Piece Value table. For example, the evaluation for the move *Nf3−e5* is the value of a knight on *e5* minus the value of a knight on *f3*. The only complicated part of this function is updating *PawnBit*. *StateValu* also handles castling bonus and the bonus for exchanging pieces when ahead.

SEARCH.CH

The *SEARCH* module, found in the SEARCH file, is the largest, most important, and most complicated of the modules.

At the bottom of the code is the *FindMove* procedure, which contains initialization data. *FindMove* calculates the Piece Value table and performs the iterative search loop with the Alpha-Beta window. If *Level* is equal to *MateSearch*, the program solves mate problems. *MaxDepth* contains the search depth of the current iteration, *MainLine* contains the best line found so far and *MainValu* contains the evaluation for the position, which is usually the same as the evaluation of the *MainLine*. *TimeUsed* tests whether or not all the search time has been used. *CallSearch* calls the *Search* function with the correct parameters.

The SEARCH module is filled with small details and at times seems a bit messy. Most other large programs—for example, compilers— can be split very nicely into well-defined and well-structured parts, but this does not really work with chess programs.

The *Search* function performs an analysis of the possible moves. We've covered this in the section on Alpha-Beta and the organized way moves are created and presented to the search. Search consists of:

- a loop that handles moves one by one
- a move generator that generates the moves one by one
- some initialization and updating performed outside the loop

Because of implementation details, the real *Search* function looks a bit different than our ideal. This is because it is difficult to write a move generator in Pascal that generates moves one by one. The body of the loop is contained in a separate local procedure, called *LoopBody*. The *Search* function calls the move generator (procedure *SearchMovGen*), which generates all the moves and calls the *LoopBody* procedure for each move. The *LoopBody* procedure calls the

Search function recursively. *LoopBody* can be regarded as the loop body of a loop in the *Search* function, and not as a separate procedure. When a cut-off occurs, the program jumps out of the move generation loop—from the *LoopBody* procedure to the surrounding *Search* function.

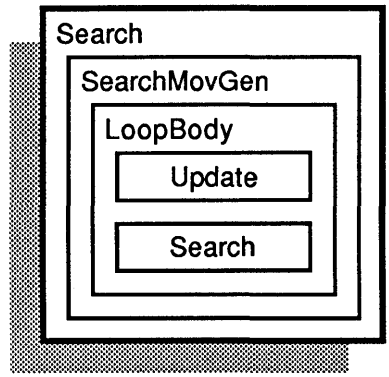


Figure 6-16. Structure of the Search Procedure

On exit, *Search* returns the evaluation of the position. The parameters are *Alpha, Beta* and *Ply* (also described in the search algorithm section), an *Inf* record, which provides some additional information about the evaluation and principal variation search, and *BestLine*, which on exit gets printed in the evaluation panel of the screen and contains the best line found in the search. *Player* contains the next player to move, *Depth* contains the depth of the position in the tree; in the first *Search*, call *Depth* is 0. The moves generated by the move generator are placed in *MovTab[Depth]*.

The body of the *Search* function first checks whether it should perform a capture search or normal search, and initializes *MaxVal*. Then it performs the search loop, which we will look at in a moment. If no legal moves are found, it checks whether the player is in check in order to determine whether the position is a stalemate or a checkmate. Finally, the *KillingMove* table is updated with the best move found in the search, which is either *the* best move or a move that caused a cut-off. This updating is done by the *UpdateKill* procedure, found at the top of the *Search* function. The killing move algorithm is described in the Program Design section on page 63.

The move generator (*SearchMovGen*) is nearly identical to the one in the MOVGEN module, except that it sorts the moves. (See the description for the *MoveGenerator* on page 61.

Even small changes to the update function can cause dramatic changes to the whole program. There is much disagreement among chess programmers about how a search should be controlled. and whether or not to use selection. If you want to experiment with the program, this is the best place to do it. The key to making good chess programs lies somewhere within these two pages of code—all you have to do is find it!

Now let's look at the *LoopBody*, starting with the body itself. *Line* is used to calculate *BestLine*. *PrincipVar*, *ZeroWindow* and the label *RepeatSearch* implement the Principal Variation Search (described in the search algorithm section on page 57). The *Update* function performs the move and updates various data structures, including the evaluation. If the move is illegal or if selection is implemented, *Update* determines that the move should not be analyzed. In that case, *Update* returns the value *TRUE*, and *LoopBody* skips the move and returns to the move generator. The *Draw* function checks whether the game is a draw because of third repetition, and calculates draw bonuses/penalties (as described in the evaluation function section on page 48). Next, there is a recursive call to *Search*. When *Search* returns, the move is taken back.

The next part of the code scans the keyboard, communicates with the user and sometimes skips the search. The program analyzes during the opponent's time of reflection, so it must be able to read from the keyboard and communicate with the user while it is analyzing. Since parallel processing is impossible in Pascal, the program must simulate it by scanning the keyboard all the time, and if the user types something it must stop the search, communicate with the user and resume the search afterwards. Communicating with the user could involve skipping the search, entering a move, changing a position, etc. Sometimes the result of the communication is that the search must be skipped. The search is also skipped if too much time has been used. Finally, *MaxVal* and *BestLine* are updated, and cut-off is eventually performed.

Now we have reached the heart of the program, the *Update* function. The *Update* function determines whether or not—and to what extent—a move should be analyzed. It actually controls the whole search, and is therefore an extremely important function.

The idea of not counting some moves as a full ply can be expanded. You could count some moves as half a ply, but you should be very careful not to blow up the search. Much of the extra time used is often completely wasted. Since the program analyzes all possible moves, you can be absolutely sure that if there is a way to avoid making the right move, the program will find it.

Let's look at the *Update* function line by line. *NextPly* is the ply depth used in the next call to *Search*, usually equal to *Ply* − *1*. We'll skip the part about *MateSearch*, since it is only used for solving mate problems. The next part of the code is about a special limited capture search that speeds up 1-ply searches; this allows the program to play "blitz chess." Next, the static evaluation function is calculated using *StateValu*. This value is placed in *Next.Value*. Many programs contain a *dynamic* part of the evaluation function which is calculated for each position. For example, a bonus might be awarded for moves that threaten two or more of the opponent's pieces. This allows the program to find combination moves one ply earlier. The total evaluation for the position is then placed in *Next.Evaluation*. In this program, however, *Next.Evaluation* is always equal to *Next.Value*.

Next, *Update* determines whether the move is a check—that is, whether the moving piece is putting the opponent in check. Discovered checks are not considered checks. This information is placed in *CheckTab[Depth]*. Moves that give check are not counted as a ply in the search. This is implemented simply by setting *NextPly* equal to *Ply*. Next, the variable *PassedPawn[Depth]* is calculated. It contains the square of a pawn that may eventually reach seventh rank; this is used in the move generation. Moving a pawn to the seventh rank is not counted as a ply either.

Although selection can speed up a program tremendously, it is very unreliable. If you want *reliable* programs, don't use selection. It is a good idea, though, to use *some* selection in the capture search. A program spends most of its time on capture searches, and if you use singlestep mode, you'll see that most of the capture search is complete nonsense. It is possible to find algorithms that will avoid most of the nonsense moves in the capture search.

If you want to experiment with selection, this is the place to do it. A *selective program* is a program that performs selection at higher nodes in the tree. A small function called *Cut* makes the selection. The program estimates the highest possible evaluation of the position, and passes this estimate as a parameter to *Cut*. *Cut* updates *MaxVal* and returns TRUE if the highest possible evaluation is not high enough (lower or equal to *Alpha*). Since selection is performed only at endnodes, the highest possible evaluation is simply *Next.Value*. A variable *Selection* determines whether or not selection is performed. Selection is only performed at endnodes and in the capture search, and never at ply 1; selection at ply 1 would make it difficult to count the number of legal moves. So Turbo Chess is still a brute force program.

Finally, the program makes a move and checks whether it is legal. *LegalMoves* counts the number of legal moves at ply one; if there is only one, the move can be played immediately. A random value of four positional points is used for some variation in the play.

TALK.CH and SMALL.CH

One of the major advances in modern interactive games is the computer's ability to analyze while the human player is preparing to move. Turbo Chess takes advantage of this optimization in the interaction of its TALK and SMALL modules.

The TALK module contains the main loop of the program, and uses the *ThinkAwhile* procedure to analyze the current game position and search for moves during the opponent's time of reflection. When a key is pressed, the *SmallTalk* procedure in the SMALL module is called.

SmallTalk handles move entries. When *SmallTalk* is called, the global variable *Command* contains the keyboard entry. If the command is a move, *SmallTalk* checks to see if the move is legal, and if it is, makes the move and then sets *Command* to *Play*, which tells the TALK module to make the counter move. *SmallTalk* also handles the following user options: Quit, MultiMove, SingleStep, AutoPlay and Hint commands. If *Command* contains one of these options, *SmallTalk* will handle the option and then *Command*. Otherwise, it does noth-

ing and *Command* is returned unchanged, and the command is handled by the TALK module.

TALK handles the following options from the main menu: NewGame, Level, Hint, Back, Forward, Turn, Edit, Value and Play. These commands are executed by the TALK procedures that follow:

- *ReadMove* gets the user input and calls *SmallTalk* to see if it can handle the command.
- *CheckMove* handles the user commands not taken care of by *SmallTalk*. It updates the appropriate global information and prints the "new" board for the Back, Forward and Turn options.
- The *ProgramMove* sub-program makes a move for the program by calling *FindOpeningMove* if the move is contained in the opening library. Otherwise, the *FindMove* procedure (in the SEARCH module) is called to make the move. *ProgramMove* executes the Play option.
- The *Edit* procedure allows the user to modify the chess board by entering the positions of each piece or by loading the board from a disk file by using the *LoadIt* procedure. The current board can then be saved to a disk file with the *SaveIt* procedure (in the INPUT module). The *Edit* procedure also handles the Edit option.
- The *SetLevel* procedure sets the game playing level any time during the game. It also handles the Level option.
- *DisplayPVTable* displays the Piece Value table on the screen for the piece specified by the user. It also handles the Value option.
- *StartUp* initializes game variables and opens the CHESS output file.
- *NewGame* reinitializes variables necessary to start a new game and puts a new game header in the output file. It also handles the NewGame option.
- *ResetGame* (called by *NewGame* and *StartUp*) starts the game and sets up the screen.
- *FindHintMove* handles the Hint option.

Some Final Comments

By now you know how a chess program works, and maybe you have even modified the program provided on your distribution disk and made it your own personal chess program.

If you want to keep up with the latest in computer chess algorithms, you can become a member of ICCA (International Computer Chess Association). ICCA is an association for chess programmers and others interested in computer chess, and has about 500 members around the world. It publishes about four papers each year, written by its members. Most of the articles are about computer chess algorithms and computer chess events; there is little written about commercial chess computers. To become a member, send 15 U.S. dollars to:

ICCA c/o Mr. W. Blanchard
Mid-America Federal Saving Bank
No. 271971560
Naperville, IL 60540
Account no. 600132099

Readers from the rest of the world should write to:

ICCA – Europe
c/o Dr. H.J. van den Herik
AMRO-Bank no. 450790878
Mekelweg/Christiaan Huygensweg
Postbus 300, 2600 AH DELFT
The Netherlands
Postgiro account no. 460175

Will a computer ever become world champion? Computers have already shown that they can play chess as well as the best human players. Some people don't believe artificial intelligence can work because they don't believe computers can *think*. Computer chess has shown that computers can perform very complicated tasks without thinking—which is something we can say about many of us, too.

Chapter 7
BRIDGE PROGRAM DESIGN

This chapter leads you through the Turbo Bridge source code. We discuss the play algorithms and bidding system used by Turbo Bridge, so you'll be able to modify the source code or try your hand at your own bridge games.

First, we investigate some of the issues involved in bridge program design: bidding, playing the cards, and evaluating both. In the last part of the chapter, we take a detailed look at each module in the Turbo Bridge program.

It is assumed that you have some understanding of the game of bridge. If you need more information, see Appendix C, or refer to the books listed in Appendix D, "Suggested Reading." If you run across bridge terms you don't understand, check the Glossary in Appendix E.

The Challenge of Bridge Program Design

The algorithms used in the chess program are the result of more than ten years of research by computer experts, and are used today because they have proven to be the best algorithms available. The algorithms used in Turbo Bridge have no such pedigree. It is therefore important that you consider them with some skepticism. They are very simple ideas; you may be able to improve the program substantially.

Designing a good bridge program is much more difficult than designing a good chess program. In bridge, there is more *unknown information* (the hands of the other players), and you have to *communicate* properly with your partner. Unknown information makes everything much more complicated. Communication isn't easy either; it is not enough to tell about your own hand—you also have to know what your partner knows about your hand. The worst problem is that you must look much further ahead in bridge than in chess. You can play good chess by looking only two to four half-moves ahead. Good human bridge players, however, always make a plan for the entire game at the outset; a good program should do the same. In chess, a simple full-width search gives good results. But if you want to look 13 tricks ahead, a full-width search is out of the question. If you look only at the different possible leads (and ignore the problem of unknown information), there are $13 \times 12 \times 11 \times \ldots \times 2 \times 1 = 6{,}227{,}020{,}800$ possibilities for tricks!

The Easy Part: the Bidding Algorithm

Programming a good bridge *bidder* is far easier than programming a good bridge *player*. For this reason, we simplify our approach in the Turbo Bridge program by keeping the routines that control bidding and playing the cards separate from each other. Most bids are predetermined by the rules of the bidding system (described later). There are a few situations for which the program will have to make a choice, but we can handle those situations. In nearly all bridge programs bidding is therefore much more consistent than the play.

The Turbo Bridge program is valuable to new or inexperienced players. The program makes legal and generally reasonable bids and plays, giving the user the opportunity to "play bridge" against real competition without having to gather three other people. The format is easy to follow, and allows the user to start playing right away.

Of interest to the experienced programmer or the better bridge player is the capability of modifying and improving the program. There has never been a bridge program written which plays at the expert level, and the ambitious user will enjoy tackling that challenge, with the format already structured in the program.

Kit Woolsey

Simple algorithms do the bidding in Turbo Bridge. Because the bidding system is isolated from the rest of the program, and because it closely follows a defined set of bidding rules, you may wish to modify this part of the code. Changing the bidding system (or implementing a completely new one) should be relatively easy.

Bidding consists of three tasks:

- You must understand your partner's bids
- You must understand what your opponents are up to
- You must determine a bid yourself

In most bidding systems, a bid usually reveals something about the length of a suit and the strength of the hand. Turbo Bridge is designed for this type of bidding.

Based on a player's bid, the program stores the information the player has revealed about his or her hand. The stored information is the *minimum length of each suit* and the *minimum and maximum number of points*. These are the only types of information the program is able to understand. The stored information is updated each time a player bids, and in this way, the program knows and understands what each player has revealed. The stored information is, of course, also known by all players (which also helps players keep track of what they have themselves revealed).

Theoretically, determining what a bid reveals should be fairly simple. For example, an opening bid of one in a suit suggests that the bidder holds 13–23 points and at least four cards in the suit. In a real game of bridge, however, bidding is almost never this predictable (and is therefore a lot more interesting).

To further complicate the programming, most bidding systems recognize different *classes* (or types) of bids. In the standard Goren bidding system, for instance, there are bid classes such as opening one in a suit, opening two in a suit, responses to opening one in a suit, etc. When determining what a certain bid reveals about a player's hand, the program first determines what bid class the bid belongs to (this depends on the previous bids). After that, it determines the meaning of the bid in the particular situation.

Turbo Bridge divides its bidding system (modified Goren) into 26 different bid classes. The bidding classes are shown below.

When you program the *BidClass* function, you must be very careful to think of every possible situation that can occur. You might tell the program that passing to an opening two in suit is an illegal bid. But if you forget to tell the program that this is not the case if the opponent overcalls, then the results can be disastrous. When humans create bidding systems, there are often some situations which they forget. This is usually not catastrophic, because the two players are able to understand each other reasonably well if such a situation occurs. But with bridge programs, you can be quite sure that the result will be a disaster (as you know, Murphy's law—"If anything can go wrong, it *will* go wrong"—is the most important law in computer science, and especially in artificial intelligence).

Turbo Bridge Bidding Classes

Bid	**Example Bidding Sequence**
(Opening Bids)	
Pass	Pass
1 in suit	1H
2 in suit	2H
Notrump bids	1 NT
(Responses to Opening 1 in Suit)	
Pass	1H, Pass
Raise to 2 in partner's suit	1H, 2H
Raise to 3 in partner's suit	1H, 3H
New suit at 1 level	1H, 1S
New suit at 2 level	1H, 2C
1 NT	1H, 1 NT
2 NT	1H, 2 NT
(Responses to Opening 2 in Suit)	
2 NT	2H, 2 NT
Normal	2H, others
(Conventions)	
Stayman	1 NT, 2C
Response to Stayman	2H
Blackwood	4 NT
Response to Blackwood	5C
(Other Bids)	
Response 2 in suit to 1 NT	1 NT, 2H
Opening 1 in suit, second bid NT	1H, 2C, 2 NT
Shutout bids	1H, 2S or 3H
Overcall bids from opponent	1H, 1S inclusive
(Normal Bids)	
Pass	
D	
R	
Natural bids	
Illegal bids	

Determining the Bid Class

It is not difficult to determine the class of a given bid. A *pass* is an *opening pass* if neither the player nor the partner has bid anything other than *pass*. The opening bid and the partner's response usually belong to special classes, while most other bids are *natural bids*. The *illegal bids* are bids that—according to our bidding system—should not be made (such as passing to an opening bid of two in a suit), usually because they don't make sense in the context of the bids made so far.

Determining the class of a bid is actually a kind of lexical analysis performed by our program. The program passes a bid value as a parameter to the *BidClass* function; *BidClass* then returns the class.

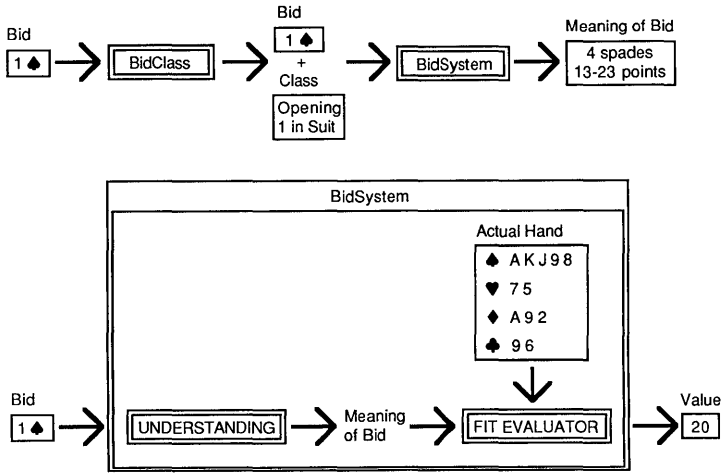


Figure 7-1. How BidClass and BidSystem Work

Using the same code for both making and understanding the bid has a number of advantages:

- the code is only written once
- the code only appears once (so keyboard mistakes are kept to a minimum and source code is smaller)
- both the bidder and the partner (either machine or human) can use a consistent system

Code takes up more memory than parameter-passing in most cases. If a passed parameter can make one procedure stand for two—or more—procedures, it's a good way to design your program.

When the class of the bid is determined, the exact meaning of the bid is calculated by the *BidSystem* procedure. *BidSystem* consists of one **case** statement, with one entry for each bid class. For each bid class, *BidSystem* defines what the bid reveals (in terms of minimum length of suits, and minimum and maximum number of points). For example, a bid of the class *opening 1 in suit* suggest that the player holds between 13 and 23 points, and at least four cards in the suit. For most bid classes, determining what the bid reveals is a trivial problem, but for the more general classes like *natural bids*, *BidClass* must be more intelligent. Each time a bid is made, the meaning of the bid is calculated and used to update the information about what each player has revealed.

The second task of the bidding program is to find a bid for a player. To do this, the program uses *BidClass* and *BidSystem* again. Turbo Bridge needs only one representation of the bidding system (instead of two). In this way, the representation is never inconsistent, and it is therefore impossible for two program players to misunderstand each other (since the same piece of code makes and understands the bid).

An evaluation function in *BidSystem* determines which bid to make, and evaluates how well the meaning of the bid fits with the player's hand. The function evaluates all possible bids, and chooses the one with the highest value.

BidSystem, then, performs two different (but related) tasks: it calculates the meaning of a bid, and evaluates how well a prospective bid fits with the hand a particular player holds. The complexity involved in evaluating bids comes from the information that a partner's bid carries about the cards in his or her hand. The evaluation of the *second* partner's next bid changes according to what the *first* partner bids.

BidClass and *BidSystem* comprise only 200 lines of code. You can change bidding conventions or implement an entirely new bidding system by altering just these two procedures.

The Hard Part: the Play Algorithm

The play algorithm for a bridge program is by far the toughest code to write. Theoretically, the program could use an evaluation procedure like the one in Go-Moku to determine which card a player should play during any particular trick. In practice, this is impractical, since the outcome of any trick affects how later tricks are played. As we said earlier, good human players make a plan for the entire game *from the outset.*

Simplifying the Problem

To start with, let's assume we are going to play with open cards (double dummy). This partially avoids the problem of *unknown information* (mentioned earlier in this chapter). Given open cards, the program can perform a full-width search 13 tricks ahead. A full-width search of a 13-trick hand would require between 1 and 100 *million* nodes. A typical PC can search *50* nodes per second. This approach is good only if we don't care about realtime responsiveness—not appropriate for an interactive game.

Consequently, the program must restrict the search in some way; either by not searching all 13 tricks ahead or by searching selectively—not trying all the possibilities at each node. It is almost impossible to program a reliable evaluation function. To make a reliable evaluation, we would have to settle on a rigid game plan at the outset,

which wouldn't take into account changes in strategy later in the game. Selection is a far more promising strategy in this case.

Assuming we can design a program to play well with open cards, how do we create a program that will work with hidden hands?

Since we are in the play portion of the bridge game, we have some knowledge of card distribution obtained from bidding. But it is unlikely that the program will be able to locate every card of each suit from the bidding (if it can, then the program plays as though all the cards were open). There are a number of cards that the program cannot definitely place.

We could tell the program to distribute the unknown cards systematically in various hands, and then "silently" play hypothetical tricks for each of the possible distributions. For each card in each distribution, the program analyzes the result by playing the rest of the game with "open" cards. Then it plays the card that gives the best average result in the different distributions. The algorithm looks like this when written in Pascal-like "pseudocode":

```
procedure FindCard;
begin
  Value[all cards] := 0;
  for DealNo := 1 to MaxDeal do
  begin
    DealCards;
    for Card :  each possible card do
    begin
      PlayCard(Card);
      Value[Card] := Value[Card] + Analyze; {Analyze calculates points for number}
      TakeBackCard(Card);          { of tricks won and bonus for a made contract }
    end;                                        { So we can try next possibility   }
  end;
    BestChoice := card with highest combined value;
end;
```

This algorithm is used in Turbo Bridge. If the number of different deals is large enough, and the analysis with open cards is reliable enough, this algorithm *should* automatically be able to perform safety plays, finesses, cross-ruffs, etc.

No algorithm is perfect, however, and Turbo Bridge is meant to be taken apart and tinkered with. As it stands, the playing algorithm "expects" that all players can make a reasonable guess about the

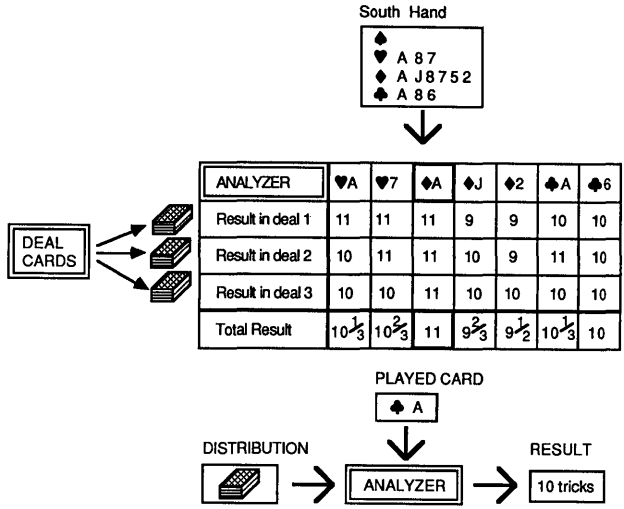


Figure 7-2. How the Analysis Algorithm Works

unknown hands. But this is not *always* the case. Also, unlike human players, the algorithm will never try to fool the opponent by making illogical or deceptive card selections. Most importantly, if the playing algorithm can make a finesse in two different directions, it tends to postpone the finesse—because the next trick will tell it which way the finesse should have been made.

Playing a Card

To successfully implement the selective search, the program must have a procedure to select which cards should be tried in the search for the proper card to play. This procedure must be very reliable, since the program will make inept plays if the correct card isn't chosen. This means that much game status information must be taken into account and analyzed, which slows the procedure down considerably. The search must therefore be very narrow.

Using a search tree, the program analyzes about 50 nodes (played cards) per second on a 5 MHz IBM PC. Analyzing the opening lead in one deal with open cards, 13 tricks ahead, and without any *branching* in the tree, requires 13*52 = 676 nodes. (Branching is the fol-

The weighting between the search results and the heuristic selection is determined by the constants *SearchFac* and *HeurisFac*. If you set *SearchFac* to 0, the program will perform no search at all, and will instead play entirely with the heuristics.

lowing of alternate paths depending on unknown opponent factors.) There are 13 different possible opening leads, and each one must be analyzed 13 tricks or 52 cards ahead. The search therefore has to be extremely narrow. The program uses no branching at all in the tree, and it tries only three different random deals. This gives a maximum of 3*676 = 2028 nodes or about 40 seconds for calculating a lead. Search factors are controlled by two constants: *MaxDeals* determines the number of different deals, and *BranchValue* controls the branching of the tree. A value of 0 for *BranchValue* (default) means no branching at all, while a value between 1 and 20 results in some branching. If you change *BranchValue* to a higher value, the program will only use branching in the leads; for the other three cards in the trick, only one card is selected to be played.

The traversal through the search tree is controlled by a special version of the *Alpha-Beta algorithm*. The result of an analysis is simply the number of tricks won by the declarer, with an extra bonus if s/he wins the contract. At the end, this algorithm chooses the card with the highest average result. If several different cards are equally good, heuristics is used to determine which card to play.

Program Structure

The Turbo Bridge program consists of about 5000 lines of code. Please refer to the source code as we examine the program structure. The following sections describe the main BRIDGE file and the include files in the order they are included in the main program: DISPLAY, SCORE, DEFAULTS, INIT, INPUT, BID and PLAY. Include files have the extension .BR.

BRIDGE

BRIDGE.PAS contains the most important global data types, data structures, the include modules, and the program body.

Data Types

At the top of the file are some constants for controlling the search (*SearchFac* and *HeurisFac*). Next are the data types that represent *hands*, *cards*, *bids*, etc. Because the type *Hands* is enumerated as *North*, *East*, *South* and *West*, the program can also refer to them using the values of 0 to 3. (For more information about enumerated data types, consult the *Turbo Pascal* or *Turbo Tutor* reference manuals.) A *suit* is enumerated as *Club, Diamond, Heart* or *Spade*, and a *trump* is either a suit or *Nt* (notrump). The card *values* are numbers between 2 and 14 (14 being ace). The programmatic description of a *card* consists of a *Suit* and a *Value* plus two booleans, *Known* and *Played*, used when playing the cards. *Played* indicates whether the card has been played yet. *Known* indicates whether the card is known by the

current player (the one who should play the next card). Finally, a bid consists of a *Trump* suit and a *Level*, which is a number between 0 and 7. The 0 level represents the three special bids, *Pass, DBL* and *RDBL*.

Program Body

After several initialization procedures, the play begins as outlined in the source code below, and continues until the user exits from the program.

```
program Bridge;
    .
    .
    .
var
  BestBid    : BidType;                                  { Chosen bid  }
  BestChoice : CardNoType;                               { Chosen card }
  Redeal : boolean.
begin                                                   { Program body }
  InitDefaults;                                { Default menu: hands played by user;}
                                               { hands displayed;program to cheat   }
  InitGames;
  InitBids;
  InitScore;
  NewScreen                                          { Screen initialization }
  repeat
    ResetGame;
    while not DoneBidding(BestBid) do;                        { Do bidding }
    CleanUpBids;                                    { Get ready to play }
    if Contract.Level > 0 then
    begin
      DummyMessage;                          { Play for your partner the declarer? }
      StartPlay;
      Play Cards (Best Choice, Redeal);
      ResetPartner;                                { if you switched with declarer }
      if not Redeal then                                 { if game not cancelled }
      begin
        PrintResult;
        CalculateScore;
      end;
    end;                                                                  { if }
    else                                                        { Everyone passed! }
      ChangeDefaults;                                  { Redisplay defaults menu? }
    until false;                                     { Program terminates when user  }
end.                                                 { selects eXit on the main menu }
```

How does the program represent a whole distribution of cards? Just as in the chess program, the optimal representation depends on what we want to do with the distribution. For evaluation and selection purposes, we want the program to be able to play and unplay cards quickly. For example, the program will often need information such as the highest card in a suit still in the player's hand, or the number of cards in a suit. But the program doesn't need information about which particular hand holds a particular card. Most importantly, we want a simple data structure so that the program will be easy to understand.

The BRIDGE file contains the main program (the top level of the program). It shows the structure of the program in 25 lines of code. In a way, the main program is a *table of contents* to the rest of the program. Organizing programs in this way usually makes it a lot easier to understand the structure quickly.

The program represents a *distribution* as an **array** [HandType,0..12] **of** CardType. Each suit is sorted with the highest card first (this makes it faster to find the highest or lowest card in a suit). Global variables *RDist* and *SDist* hold the distribution. *RDist* always contains the actual distribution of the cards, while *SDist* contains the simulated distribution used in the search (in which unknown cards are distributed at random).

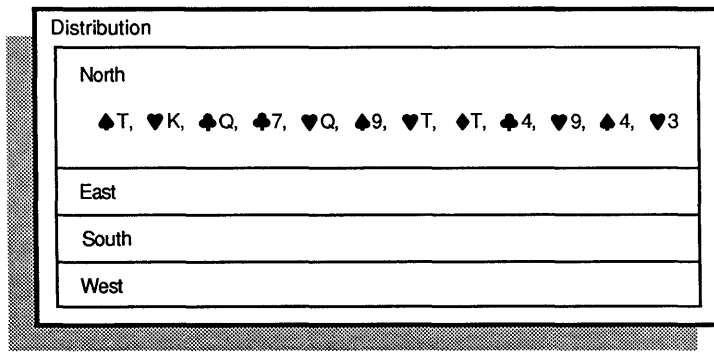


Figure 7-3. Distribution

RData and *SData* contain additional information about the distribution. For each hand, *L[Suit] contains the length of the suit, while P* contains the number of points (both high card points and distribution points that hand earns). When a card is played from the hand, the length of the suit, but not the number of points, is updated. Thus, *P* contains the number of points before any cards were played.

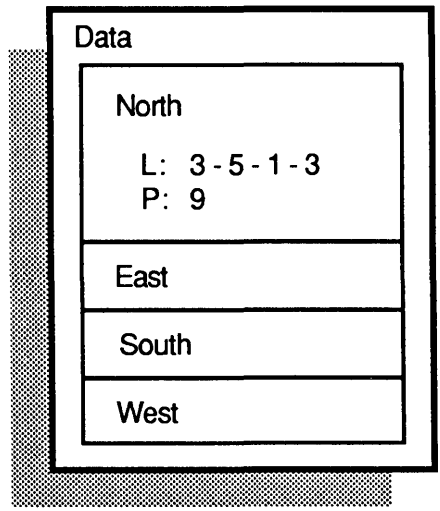


Figure 7-4. Data

Info contains the information each player has revealed during bidding and play. *Info* is what the players know about each other's hands. All players have full access to the information in *Info*. For each hand, *MinL[Suit]* contains the minimum length of the suit, while *MinP* and *MaxP* contain the minimum and maximum number of points. Notice the correspondence between *RData* and *Info*. *RData* contains actual information, while *Info* contains information that the players have revealed. Just as with *RData*, *MinL* (but not *MinP* and *MaxP*) is updated when a card is played from the hand. If *MinL* of a suit is equal to -1, it means that the player has revealed a void suit in the play. If *MinL* of a suit is equal to 0, it means that the minimum length is 0, and players know nothing at all about the suit.

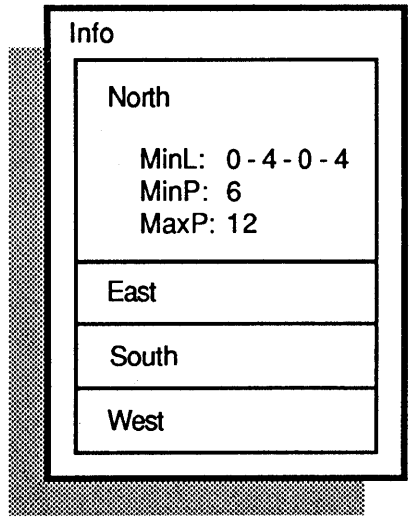


Figure 7-5. Info

Apart from the distribution of the cards, the program must also keep track of bids and played cards. *Bids* is an **array of** *BidType*, and contains all the bids made. *BidNo* contains the number of bids made. *Bids[BidNo-1]* always contains the bid immediately before the current one. *Dealer* contains the first player to bid in the game.

Contract contains the contract, *Doubled* contains the doubling status (undoubled, doubled or redoubled) and *Dummy* contains the partner of the declarer.

The played cards are stored in the *Game* array. *Round* always contains the number of played cards, and thus *Game[Round-1]* always contains the most recently played card. Instead of containing the cards themselves, *Game* contains the *Hand* and the number (*No*) of the card. The card itself is then found in *RDist[Hand,No]*.

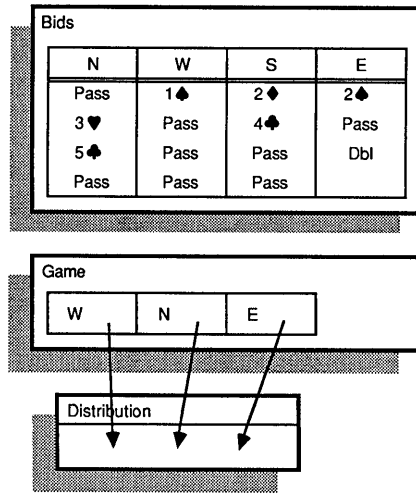


Figure 7-6. Bids and Game

Turbo Bridge needs more information to store the full game status during play. The *Rel* and *Sim* records hold the current status of the game—the current round, the number of won tricks for each side, and the current trick. These records correspond to the actual and simulated games, just as *RDist* and *SDist* do. *Round* contains the number of played cards. *TrickNo* contains the number of played tricks (equal to *Round DIV 4*). *WonTricks* contains the number of tricks won by the declarer. *LeadHand* contains the player who led the first card in the current trick, and *LeadSuit* contains the suit of that card. *BestCard* contains the best card in the current trick, and *BestHand* contains the hand which played the card. Finally, *PlayingHand* contains the hand which will play the next card.

Turbo Bridge must store information to correctly keep the score. This data must be declared globally so that the score can be maintained between games and rubbers. The *Games* record contains the value of the current game (1..3) and the scores of both teams in each of the respective games. Below is the type and variable information used by the scoring routines.

```
type
  GameRecordType = array[1..3] of integer;    { Underline score for the 3 games }
  ScoreType = record                                          { Holds team score }
                GamesWon,
                OverLine,                                        { Bonus points }
                Total      : integer;                          { Overall totals }
                GameRecord :  GameRecordType;                  { Underline scores }
              end;
  GameTableType = record
                    CurGame : integer;                       { Current game 1..3 }
                    N_S,                                      { North/South team }
                    E_W     : ScoreType;                        { East/West team }
                  end;
var
  Games : GameTableType;                { Holds the global scoring information }
```

DISPLAY.BR

The DISPLAY module contains all the procedures for writing on the screen, writing messages to the user, etc. If you wish to try the program on a different computer or if you want to adapt the screen picture to a high-resolution graphics monitor that will show pictures of the cards, or use windowing or mouse-control, this is the part of the code you need to change. Borland's *Turbo Graphix Toolbox* will help you experiment with graphics displays.

First the module defines the strings used to display *HandNames, TrumpNames* and *ValueNames*. *TrumpSQS* contains special characters for club, diamond, heart and spade. The IBM PC character set assigns special characters—among them the symbols for card suits—to the character numbers between 128 and 255.

The procedure *EqBQS* checks whether two bids are equal. *DistPoints* calculates the number of distribution points for a hand (these two general functions are placed near the top of the *Display* procedure because they are used throughout the DISPLAY module but nowhere else in the program).

BidStr and *CardStr* convert a bid or a card to a character string. Just as in the chess program, the program stores the played games on the disk. The program uses the special suit characters whenever it displays a bid or a card on the screen. Not all printers are capable of producing these characters. Because of this, the program uses ASCII characters when storing to disk. To make this conversion, the program uses the functions *AsciiBidStr* and *AsciiCardStr*, which work like *BidStr* and *CardStr*.

Frame is a small procedure that draws the frame around the bridge table in the middle of the screen.

TopHand contains the hand placed at the top of the screen picture (north during bidding, and the dummy during play). Each hand is displayed in a separate "window" on the screen. The *GoToHand*

procedure is very similar to the standard Turbo Pascal *GoToXY* procedure, but is given a *Hand* as parameter and places the cursor within the "window" of the specified hand. Similarly *GoToCardPos* places the cursor at the card position on the bridge table in the middle of the screen.

WindowHand contains the hand placed at the left column of the information window (at the lower left corner of the screen). This hand will belong to the dealer.

PrintNames is given a *Hand* as parameter, resets the information window, and prints the hand names with the correct hand in the leftmost column. *PrintInfo* is given a *Hand* and a string as parameters, and displays the string at the correct place both on the bridge table and in the information window. The rest of the program need not "worry" about the information window at all. *OutputNames* and *OutputInfo* work in exactly the same way, but output the information to disk.

All these procedures are used by *PrintBid* and *PrintCard*, which make sure that the bids and the played cards are printed and erased at the appropriate time and place.

HandKnown checks whether a hand should be displayed on the screen. The program displays hands if you've requested it during initialization, or if a particular player becomes the dummy. *PrintScreen* sets up the screen picture with the specified hand at the top of the screen.

BidMenu draws a menu of user commands during bidding, and *PlayMenu* draws a menu of user commands during playing. *Error* displays an error message and rings a bell.

The DISPLAY module is structured with the name and color definitions at the top of the file, followed by a number of small procedures, each taking care of a subtask. This makes it a lot easier to add new features to the program.

SCORE.BR

The SCORE module contains a number of procedures that calculate and display scoring.

The *InitScore* procedure initializes the fields in the *Games* scoring record. (The *Games* scoring record is described in the *Info* section.)

The *DisplayScore* procedure contains the routines that draw the game score board and fill the score. Below is a sample game board with scoring fields, with explanations set off in comments :

BRIDGE SCORE		
{Teams}	N/S	E/W
{Overline Totals}	270	400
{Underline total game 1}	120	20
{Underline total game 2}	0	120
{Underline total game 3}	0	0
{Overall} Totals	390	540

The *CalculateScore* procedure contains all of the constant values needed to keep score—the points necessary to win a game, the score for contracts made in each of the respective suits, the bonuses for a small and a grand slam, the bonus for winning a rubber in two games, etc. To make changes to the bridge scoring system (for example, to include honor points in the scoring), the *CalculateScore* procedure should be modified.

CalculateScore uses the following routines :

- *CalculatePoints* calculates the points for the current game and updates the team's total scores.
- *CalcVulnerabilityPnts* determines the appropriate bonus points awarded to the winning team if it was vulnerable.
- *CalcGamePnts* calculates the game winning bonuses if the team that is playing the contract wins enough points to win the current game.
- *CalcPenaltyPnts* calculates the penalty points awarded to the team that lost the bid but prevented the declaring team from making the contract.

DEFAULTS.BR

The DEFAULTS module is called at the beginning of the program and allows the user to specify how the game is to be played. The user can choose to play 0 to 4 of the hands, allow all of the hands to be displayed openly, or allow the computer to cheat and look at all of the cards (including the opponents).

This module is interesting as a programming example because it is basically a cursor control module. The *GetDefaults* procedure calls a series of I/O routines that allow the user to move around the screen with the arrow keys and toggle a default selection on or off by hitting the space bar.

INIT.BR

The INIT module contains procedures to deal the cards and set up the screen picture.

Dealing the cards is not difficult. Remember, however, that the program must update both the distribution and the data arrays. The program assumes that the cards are already distributed in some way, so all it must do to achieve random distribution is mix them a bit more. The *Exchange* procedure mixes two cards, and updates *SDist* and *SData* accordingly. *ChangeCards* moves cards around in such a way that the distribution corresponds with the information in *Info* afterwards. The *DealCards* procedure mixes the unknown cards at random. Finally, the *SortCards* procedure sorts a distribution (with the highest cards in each suit first).

InitGames, called once at the beginning of the program, sets up a starting distribution in *SDist* and *SData* (so the procedures described in the paragraph above can be used to mix the cards later). *StopGames* closes the BRIDGE file and quits the program. *PrintBidScreen* and *PrintPlayScreen* set up the screen pictures for bidding and play, respectively. These procedures call the procedures from the DISPLAY module. *ResetGame* contains the initializations performed before every game. These include dealing the cards at random and setting up the screen for the bidding. *NewDeal* re-deals the current hand and clears the previous bidding information. *Clearbids* reinitializes bid information without re-dealing. *StartPlay* contains the initialization performed when the bidding is finished. It prints the contract on the screen and calculates who should play the opening lead.

INPUT.BR

The INPUT module handles most of the communication with the player. In the bidding portion of the game, when the bidder is not the program, the *Answer* procedure is called to get keyboard input. *Answer* gets the bid by calling *GetBid*. *GetBid* reads the input by calling *ReadBid* and evaluates it by calling *ParseBid*.

During bidding, the player may ask to redeal the hand if s/he was dealt a bad hand, or alternately clear all the bids without redealing. In either case, the boolean variable *Restart* is returned TRUE, which tells the BID module to reinitialize the bidding information and thus restart the bidding.

If the hand currently being played belongs to a human player, *Answer* is called to get the play. The boolean variable *Bidding* is now FALSE, so *Answer* calls *GetPlay* to read the user option with *ReadPlay* and evaluates the play with *ParsePlay*.

During either bidding or playing, the user can call up the score. This option is handled by the INPUT module and is thus transparent to the BID and PLAY modules.

At the end of the game, after displaying the score, *ChangeDefaults* is called, which prompts the user to start a new game, reset the defaults (see the DEFAULTS module) or exit the program. *GetResponse* is then called to get the user's selection.

BID.BR

The BID module contains all procedures and data used in bidding (also see the section on the bidding algorithm on page 90).

First, the program defines all the bid classes. The *BidTyp* array fits closely with the *Bids* array, and contains the bid class of each of the previous *Bids*. The opening bid is placed in *Bids[0]* and *BidTyp[0]*. (As a small programming trick, both arrays start at index -4, and from index -4 to -1 there are four opening passes. This simplifies the *BidClass* procedure somewhat, because an initial bid will only be added if it is a non pass bid. If all four players pass, no bids are updated and the game must be restarted.)

BidTyp

N	W	S	E
Opening Pass	Opening 1 in Suit	Overcall	Support 2 in Suit
Normal Bid	Normal Pass	Normal Bid	Normal Pass
Normal Bid	Normal Pass	Normal Pass	Normal Dbl
Normal Pass	Normal Pass	Normal Pass	

Figure 7-7. BidTyp

The *Jumps* function counts the number of jumps represented by a bid. For example, bidding three clubs when the current contract is one spade is a single jump (*Jumps* would return the value 1). The *Number* function counts the number of cards a player has of a given value. For example, *Number (North, Ace)* counts the number of aces that *North*'s bidding indicates s/he has.

The *BidClass* function calculates the bid class of a bid, and keeps track of all bids and responses. For example, if the partner's last bid was a Stayman bid, the current bid is a response to the Stayman bid.

One improvement you may wish to make to the Turbo Bridge source code is an individualized bidding system. The reason Turbo Bridge uses a specialized system is because standard Goren is not very systematic (and was developed way before computers). By implementing a system that you like and use regularly in your own playing, Turbo Bridge can become your own custom bridge program.

A pass may be an opening pass, a response pass to partner's opening one in a suit, or a normal pass, depending on the type of the partner's last bid. Four NT and five NT are usually Blackwood. If the partner's last bid was an opening in NT, then a bid in clubs at the next level is Stayman, while the response two in suit is the special weak two in suit response to one NT. If you change this routine, be careful to take all possible bids and responses into account.

The *BidSystem* function determines the meaning of all possible bids. It can be called in two different ways. It can either calculate and update *Info* according to a *Bid* (when the computer does not know the hand), or it can evaluate how well a *Bid* fits together with the actual hand of the *Player* (when the computer does know the hand).

InfoPlayer contains the updated version of *Info[Player]* (and thus the meaning of the *Bid*). *BidVal* contains the evaluation of the *Bid*. If *Test* is set, *BidVal* is calculated by testing the *Player*'s hand against *InfoPlayer*. *MinTab* contains the approximate number of points (high card points and distribution points for both the player and the partner) necessary to win a contract.

BidSystem is a single **case** statement with one entry for each bid class. For most of the bid classes, the **case** statement simply determines what each particular bid reveals in terms of minimum suit length (*MinL*) and minimum and maximum number of points (*MinP* and *MaxP*). For example, an opening of one in a suit suggests between 13 and 23 points, and at least four trumps. The program calculates *BidVal* using this information. If the program determines that a bid falls within the conventions it understands, it will adjust *BidVal* directly. This makes it possible to make the conventional bids that cannot be calculated using *MinP*, *MaxP* and *MinL* (opening in NT is bad if the player has more than one doubleton, and responding with anything other than two NT to an opening of two in a suit suggests at least an ace and a king).

All the bid classes are straightforward, except for the three general bid classes with normal bids (normal pass, normal bid and normal double). The code for these bids must contain some sort of decision-making intelligence, because these bids are not strictly mechanical. The final contract is determined by normal bids, because the contract is accepted by three passes. The code for these three bid classes is in the procedures *NormalPass, NormalBid* and *NormalDBL*.

Normal Passes

The *NormalPass* procedure contains the following rules. When a player passes, it means that s/he accepts the present bid as the final contract. You or the program should only pass if you believe the present contract is the best strategic move for you and your partner—whether or not you are declarer or defender. In general, passing means that you do not believe that you and your partner should bid any higher. If your partner has the present contract, passing below game means that you do not think there is any chance for game, while passing below slam means that you do not think there is any chance for slam (passing also means that you accept the present suit as trump). If the opponent to your right did not pass, you can pass and leave the initiative to your partner without any risk. Passing in this situation means that you have a weak hand.

Normal Bids

Except for the Stayman and Blackwood conventions, all Turbo Bridge bids are "natural." This means, for example, that a bid of 2 hearts indicates an ability to make a 2 hearts contract and a Double indicates an ability to defeat the opponents contract.

Normal bids are the most complicated bids. The *MinPoints* function calculates how many points are required to bid at a specific level. *MinPoints* follows these two rules: (1) a player should only bid the number of tricks s/he thinks s/he can win, and (2) if the player and partner have been bidding a trump suit, the player should not bid a different suit unless the partner can bid the trump suit again without going too high.

If the opponent to the bidder's right did not pass, then making a normal bid (instead of pass) reveals three more points than the bid otherwise would. Bidding above game level means slam interest, and a jump always means game demand. The first bid in a suit shows a four-card suit, and each successive bid in the suit shows one extra card. If your partner has bid the suit, a support bid means that you and your partner have at least eight cards in the suit, and this determines the suit as trump suit unless you can find a better suit. Bidding in NT shows no singletons (except for three NT, often used as a chance bid when you have high cards distributed amongst more than one suit).

The minimum and maximum number of points are determined by the *MinPoints* function. If you and your partner have found a fit in major, there is no reason to try to find any other suits. If your partner has used the Blackwood convention, you should let him decide the final contract and not take any initiative yourself. Three NT is a final contract bid; you should usually pass when your partner bids three NT (unless you want to bid a slam).

Normal Double

NormalDBL doubles a contract only if it believes the contract will leave opponents at least two down. Every four high card points (both player and partner) above the first four count for one trick. If a player has four or more trumps, each card above the first three counts for one trick. The program will double a four spade contract with four trumps and sixteen high card points (including points shown by partner).

Finding and Making the Bid

After the bid is calculated using the functions and procedures described above, the new bidding information is added. If the task was to update *Info*, the bid is evaluated depending on how much new information it gives the partner. Otherwise, the bid is evaluated by comparing the information with the actual hand. A large penalty is given if they do not correspond. The different parameters were determined by trial and error. The values result in correct bids by the program in the standard situations.

FindBid finds a bid to make. If it is the program's hand, the procedure evaluates all possible bids using the evaluation function in *BidSystem* and chooses the bid with highest evaluation. If it is the human player's hand, the program reads the bid from the keyboard. The *MakeBid* procedure performs the bid by updating the global variables (*Info, Bids, Contract* etc.), and then displays the bid on the screen.

FindBid and *MakeBid* are called by *DoneBidding* until function *ThreePass* returns TRUE (i.e., there have been three consecutive passes and the contract is set). Below is the source code for the *DoneBidding* function.

```
function DoneBidding (var BestBid : BidType): boolean;
{ Successively finds and makes bid until the bidding stage of the
  game is completed by three consecutive passes or the user
  decides to restart the bidding }
var
  Restart : boolean;                      { Restart is true if the user clears }
                                            { the bids or asked a new deal    }
begin
  Restart := false;
  repeat
    FindBid(BestBid, Restart);    { If it is the program's turn to bid it finds  }
                                  { the best bid else it gets the user's bid     }
    DoneBidding := not Restart;
    if Restart then               { We must exit routine and restart bidding     }
      exit;
    MakeBid(BestBid);             { Update bidding information and print the bid }
  until ThreePass;                              { Bidding has completed normally }
end;                                                          { DoneBidding }
```

PLAY.BR

PLAY, the largest and most complicated module in Turbo Bridge, plays the cards (see the play algorithm section on page 93). *FindHigh* and *FindLow* find the highest and lowest card in a suit and return the *Index* to the card in *SDist*. These functions must never be used if the suit is void. *Highest* and *Lowest* do the same thing, but return the *Value* of the card (and also work in case of a void suit).

The *PlayCard* procedure plays a card and updates the simulation data structures (*Sim, SDist, SData*, etc.). The search uses this procedure. The corresponding take-back procedure is *ResetTrick*, which takes back a whole trick (and thus undoes four calls to *PlayCard*). The program does not need a procedure to take back a single card, since it only uses branching for the first card in each trick.

TestSuit tests the strength of the suit by calculating the number of *TopTricks* in the suit. The variable *Winning* is set if all the cards in the suit are top tricks. The function *TopTrick* also calculates the number of top tricks, but returns a negative value if the opponent has top tricks.

The heuristics for performing the card selection follow these procedures. *SelectLead* is used to select the leads, while the other three cards in the trick are selected by *SelectCard*. These functions work as move generators. *SelectLead can* be called several times in the same situation, and will give a new suggestion every time. It currently generates only one suggestion. If you change the *BranchValue* constant at the top of the program, *SelectLead* will generate up to two different suggestions. *SelectCard*, however, supplies only one suggestion (and is therefore only called once).

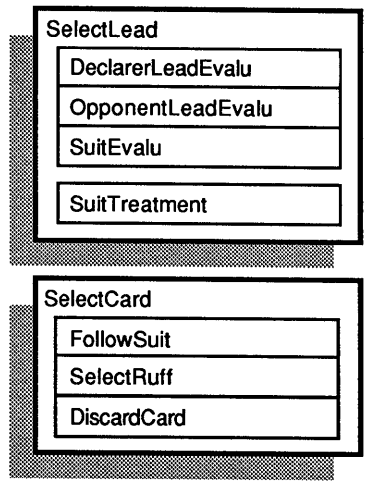


Figure 7-8. Select Lead and Select Card

When *SelectLead* is called, it first evaluates the four different suits with a simple evaluation function. The evaluation function estimates how good a lead a card in a particular suit would be. The first suggestion is made in the suit with the highest evaluation, and the second suggestion in the suit with the second highest evaluation. The evaluation function of the suits is found in the *SelectSuit* function. When the suit is determined, the *SuitTreatment* function determines which card in the suit should be played.

The *SelectSuit* function is again divided into several functions. The most important rules are found in *DeclarerLeadEvalu* and *OpponentLeadEvalu*. If these rules do not apply, the second most important rules (found in *SuitEvalu*) are used. As a special rule, the program gives a high evaluation for leading the same suit as in the last lead (providing that the reason for leading the suit is still valid). This reduces the number of analyzed cards. If the suit played the last time the player or his or her partner led a suit can be led again, and the reason why it was played last time is still valid, then this suit is usually the only suit analyzed.

The most important rules for the declarer occur in *DeclarerLeadEvalu*. First, the program tries to establish a ruff trick by playing a suit in which either the declarer or the dummy have few cards, and which cannot be ruffed by the opponents. Second, the program draws trumps until the opponents have no more trumps left.

The most important rules for the opponents occur in *OpponentLeadEvalu*. First, the program plays suits with top tricks. Second, it tries to establish a ruff trick by leading a singleton or a suit in which the partner is void or has a singleton. These rules apply only when the declarer and the dummy cannot ruff the suit.

When the most important rules do not apply to a suit, the program uses the second most important rules found in *SuitEvalu*. The declarer first plays suits with top tricks, then long suits and finally trumps. The opponents first play trumps, a reasonable and safe move, then long suits.

Once the suit is chosen by the above three procedures, the *SuitTreatment* function determines which card in the suit is to be played. Normally the lowest card is played, although the highest card may be played if both opponents have singletons or void suits, if the partner's highest card is not high enough to take the trick, or if playing the highest card is necessary to make a finesse. However, if one of the opponents has the highest card singleton, the program always plays the lowest card.

The *SelectCard* function determines which of these three functions is appropriate in the current situation: *FollowSuit* (used when the player can follow suit), *SelectRuff* (determines whether to ruff, and which trump to use) and *DiscardCard* (selects a card to discard).

The *FollowSuit* function uses different rules for the second, third and fourth hand. If the highest card in the suit is lower than the best card in the trick, the program plays the lowest card. In the second hand, the program plays low, except when playing a high card will win the trick (and the partner cannot win it) or press the third hand. In the third hand the program plays high, unless the partner is sure to win the trick. In the fourth hand, the program plays high, unless the partner already has the trick.

The *SelectRuff* function ruffs a trick if it will win the trick (and the partner cannot win it). In that case the program uses the lowest trump that will win the trick.

The *DiscardCard* function first evaluates all the suits, and then discards the lowest card in the least important suit. The evaluation function gives penalties for discarding trumps or high cards, for leaving high cards unprotected, for discarding top tricks and for discarding cards that give the opponents extra top tricks.

The heuristics are fairly simple, which is why they sometimes select the wrong card. You can study how the heuristics play alone by setting the *SearchFac* constant to 0 and allowing the program to cheat. If the heuristics were more sophisticated however, they would become much slower, and the overall strength of the program would probably not increase significantly. The only way to overcome these problems is to make the search much wider, which would reduce the importance of the selection and slow down play (although if some frequently called functions were recoded in assembly language, play could be speeded up).

The search for a card to play is performed by the *Analyze* procedure. When the analysis is finished, the number of tricks won by the declarer is placed in the *Result* variable. The *Analyze* procedure uses the Alpha-Beta algorithm, but the implementation is somewhat different from the usual one (for example, the two sides do not take turns as in chess—instead, the same side can lead several successive tricks).

The *FindCard* procedure finds the card to play. If this is the user's hand, the card is read from the keyboard. If the program plays the hand, the card is found by analying the different possibilities. Not analyzing all possible cards significantly reduces the amount of time spent in this procedure. If two cards are equally high (for example, if the same player has both the queen and the king in a suit), the procedure analyzes only one of these cards. All cards below nine are considered equally high for the purposes of analysis. If the program discards a card or follows suit with a low card, it will always play the lowest card in a suit. Exactly which cards should be tried is determined by the *SelectTry* procedure (part of the *InitSearch* procedure which initiates the search).

The heuristic evaluation is calculated by distributing the cards and giving a bonus (equal to *HeurisFac*) to the card selected by the heuristic selection functions. The search evaluation distributes the cards and analyzes the result of playing each card by using the *Analyze* procedure. The number of won tricks (multiplied by *SearchFac*) is then added to the heuristic evaluation. Winning the contract gives an extra bonus (equal to two won tricks) to the declarer. The number of different distributions is determined by *DealNo*. Finally, the card with the highest evaluation is chosen.

The last procedure in each play is *MakeCard*. This procedure plays the chosen card, displays it on the screen and updates the global variables (*Rel, RDist, RData* etc.). Below is the source code for *MakeCard*.

```
procedure MakeCard(BestChoice: CardNoType);
              { Play the Card in the real situation: and update the Rel records }
var
  PlayHand : HandType;                                          { PlayingHand }
  Card     : CardType;                                          { Played Card }
begin
  CheckKBD;                                              { Check for user abort }
  Sim := Rel;                          { Copy the records which contain the real }
  SDist := RDist;                        { situation into the simulation records }
  SData := RData;
  PlayHand := Sim.PlayingHand;
  Card := SDist[PlayHand,BestChoice];                 { Card is the best choice }
  with Sim,Card,Info[PlayHand] do
```

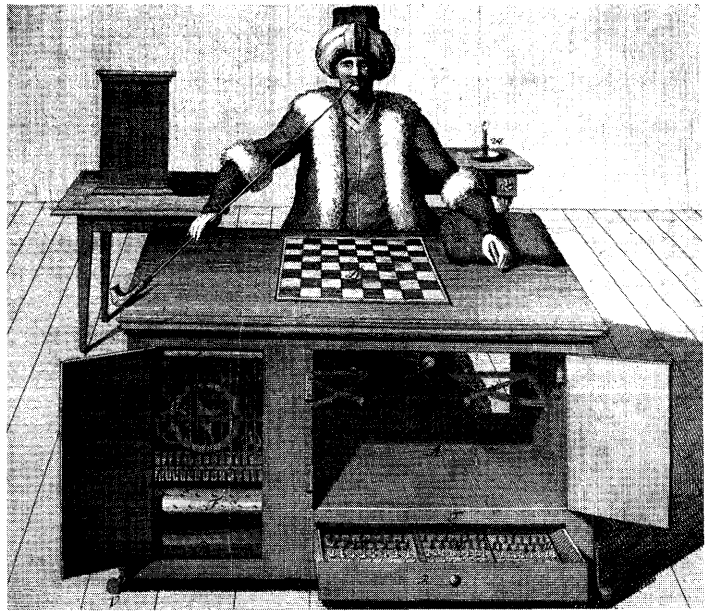
```
  begin
    if minl[suit] > 0 then                          { Card played, update info }
      MinL[Suit] := MinL[Suit]-1;
    if (Round AND 3 <> 0) AND (Suit <> LeadSuit) then
      MinL[LeadSuit] := -1; { the current hand is out of cards in the lead suit }
    CheckKBD;                                          { Check for user abort }
    PlayCard(PlayHand,BestChoice);                           { Play the Card }
    Rel := Sim;                                    { Update the "REAL" records }
    RDist := SDist;
    RData := SData;
    if Round = 1 then                        { if first round then Setup screen }
      PrintPlayScreen;
    PrintSuit(PlayHand, Card.suit);                    { Print the updated hand }
    PrintCard(PlayHand, Card);                  { Print the Card on the table }
    if (Round AND 3) = 0 then                             { The trick is done }
    begin
      PrintWonTricks;
      ClearTblMsg;
      ClearTable;
    end;
  end;
end; { MakeCard }
```

The main procedure in the PLAY module, *PlayCards*, calls *FindCard* and *MakeCard* for each of the thirteen card tricks or until the player cancels the game.

APPENDICES

Appendix A
THE HISTORY OF COMPUTER CHESS

A little historical background will help you understand the fascinating subject of computer chess.



Used by permission of the Cleveland Public Library, John G. White Collection of Folklore, Orientalia, and Chess.

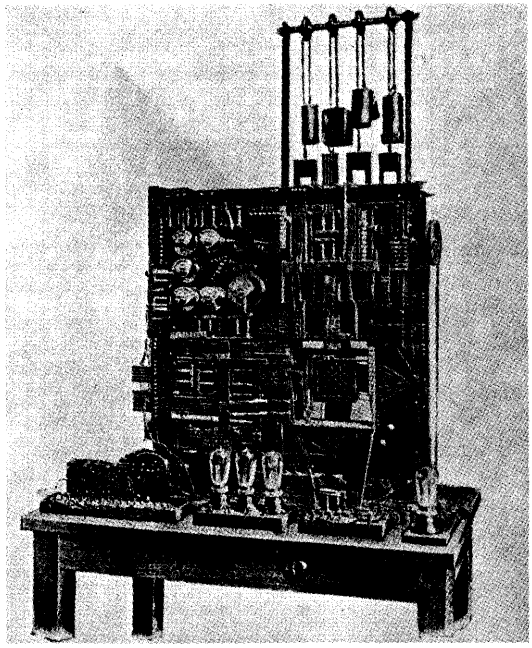
Figure A-1. The Turk

The first and still the most famous chess machine was called *The Turk*. It was built about 1770 by Baron Von Kepelen. During the 18th century, it was exhibited at all the courts of Europe. Once it beat Napoleon, who was a much better general than he was a chess player.

The Turk chess machine consisted of a large chest and a life- sized mannequin of a Turk. The Turk moved the pieces with its left hand, and played excellent chess. The reason it played so well was because of a very sophisticated system of drawers and doors that could be opened to show pulleys and cables, yet allowed a small man (a chess expert) to hide inside. Although many skeptics doubted that such a machine could play chess, the Turk and other machines like it con-

tinued to amaze people up to the end of the 19th century. The Turk was destroyed in a fire in 1854, but by reading a number of articles written about it one can reconstruct the way it worked. Although the machine was only a trick, the fact that it was created shows that people have dreamed about machines playing chess for centuries.

The first *real* chess machine was built in 1890 by a Spanish scientist, Leonardo Torres Y Quevedo. This machine could achieve mate with king and rook against king. Since this is a relatively simple way to win, it was possible to devise explicit rules and build them into a machine.



Used by permission of the Cleveland Public Library, John G. White Collection of Folklore, Orientalia, and Chess.

Figure A-2. The Torres Machine

Torres Y Quevedo built the machine to demonstrate his theories about automation. The machine was a very complex system of wires and switches (this was long before the transistor), and was one of the most sophisticated machines built at that time. It is probably *still* one of the most sophisticated computers ever built with wires and switches. It is still in good working order and can be seen in the museum at the Polytechnic in Madrid.

In 1949, Claude Shannon from Bell Labs published a paper entitled *Programming a Computer for Playing Chess*. In this paper he de-

scribed how to write a program that could play chess. This was in the very early days when computers were still quite primitive, and he wrote the paper mostly to convince people that it was possible to program computers to perform tasks normally considered to require "intelligence." What is most extraordinary about the article is that even today, most chess programs are based on the algorithms presented in this first paper on computer chess.

Computer chess through the ages
- 1770—The Turk
- 1890—Torres Machine
- 1951—Alan Turing's chess algorithms
- 1956—Los Alamos program
- 1958—Alex Bernstein
- 1966—US/USSR match
- 1967—Machack—a real tournament victory
- 1970—First computer chess tournament at ACM
- 1978—Sargon
- 1985—Turbo Chess

The first actual chess program was probably written by Alan Turing in 1951. His algorithms were too complicated to be programmed on the computers available at that time, so he had to do all the calculations by hand.

The first chess program to run on a computer was probably the *Los Alamos Program* written in 1956 (some authors believe that the Soviets won this race with a slightly earlier program, but this is not very well documented). The computers were not very large in those days, so the programmers had to restrict the game and play it on a 6×6 board, and omit the bishops. The program ran on a MANIAC computer with a speed of 11,000 instructions per second. (Today, an IBM PC performs about 500,000 instructions per second.) This program also became the first program to beat a human player.

The first full chess program was probably the one written by *Alex Bernstein* in 1958. During the next few years, many programs appeared, and computers became faster and faster chess players. In 1966, a match was arranged between a U.S. program and a Soviet program. The Soviet program smashed the opponent; the Russians have always been good at chess. The following year a program called *MacHack* by Richard Greenblatt became the first to beat a human (1510 ELO points) in a real tournament game. In 1970, the first computer chess tournament between four programs was held at the ACM conference. The winner was *Chess 3.0* by David Slate and Larry Atkins. The ACM tournament has been held annually ever since, and is today the most important computer chess tournament for large computers.

The 60's and 70's were the days of computer breakdowns, ridiculous and illegal moves, and extreme optimism among programmers. In 1968, four computer experts made a bet of nearly $3000 against David Levy, an international chess master, that within ten years he would be unable to beat the best chess program in a match. Levy won his bet in 1978 against *Chess 4.7*, a successor to *Chess 3.0*, but at the same event he also became the first International Master to be defeated by a program. If you want to know more about the earliest days of computer chess, you can read *Chess and Computers* by David Levy, Computer Science Press, 1976.

The best program written during the 70's was *Chess 4.N*. The program is still among the very best, but is now called *Nuchess*. Most of the

best programs today—including the one in this book— are based on the ideas from Nuchess. On large computers, the increase in playing strength until now has mainly been caused by increased computer power rather than better programs.

In 1980, Ken Thompson from Bell Labs built his customized *Belle* system, which has since won many chess championships. (Thompson developed the UNIX operating system, and he therefore has rather a free hand at Bell Labs.) *Belle* is a 130 pound box containing a computer and custom-built chess hardware. Because of the specialized hardware, it can analyze up to 100,000 chess positions per second. *Belle* became the first chess computer considered to be a threat to national security. When Thompson wanted to take it to a chess exhibition in Moscow, it was impounded at the airport, since U.S. authorities believed it could be of vital importance to the Soviet military.

In October of 1985, the *Cray Blitz* program, running on a Cray X-MP/48 supercomputer, was defeated at the North American computer chess championships by a computer chess machine called *HiTech*. *HiTech* was developed by a group (led by Hans Berliner) from the Computer Science department at Carnegie-Mellon University.

The machine itself is a Sun minicomputer with customized chips. It uses a program called *Oracle* to control the search strategy; once the strategy is determined, control passes to *Searcher*, a unit with 64 microprocessors (one for every square on the chessboard). When a piece is placed on a square, the appropriate chip begins evaluating probable outcomes.

The best program today is probably either *Belle* or *Cray Blitz* by Robert Hyatt. *Cray Blitz* runs on a special experimental, superfast 16-processor Cray computer. The playing strength of the best programs is about 2200 ELO. (*Chess 4.5* and *Belle* are described in *Chess Skill in Man and Machine*, edited by Peter W.Frey, second edition, Springer-Verlag, 1983.) Not everyone has access to a Cray or free reign at Bell Labs, though. In the late 70's, attention turned to developing good chess programs for personal computers.

In 1978, the first dedicated commercial chess microcomputer, *Chess Challenger 10*, was introduced in the United States. Although it did not play very well, it was a success, and it spawned an industry. A couple of companies in Hong Kong started to produce chess computers and today most dedicated chess computers are made in Hong Kong (although the best programs are still made in California!). The playing strength of microcomputer programs has increased dramatically.

The increase in microcomputer playing strength has been due primarily to better programs. It is a very good example of how much you can get out of a microcomputer if you program it correctly. Most of the commercial chess computers still use the 6502 chip (also used in the Apple II, Commodore and Atari computers). The newer 16- and 32-bit chips are faster for normal programs, but not necessarily for chess programs. For chess programs, a 4 MHz 6502 is as fast as an 8 MHz 68000! In the beginning, the microcomputer programs analyzed about 40 positions per second; today, the best commercial chess programs analyze up to 800 positions per second (running on

ACM—the Association for Computing Machinery—is located at 11 West 42nd St., New York, New York 10036. Phone: (212)869-7440.

a 4 MHz 6502). In comparison, *Cray Blitz* running on a Cray-1 computer analyzes about 15,000 positions per second, which is only 20 times as many. The CHESS.PAS program analyzes about 50 positions per second on a 5 MHz IBM PC.

If you want to see some of the programs and meet the people behind them, you can visit one of the computer chess tournaments held every year. For large computers there is the ACM tournament held at the annual ACM conference in the United States. For microcomputers there is the annual microcomputer *World Championship*, usually held in Europe. These tournaments are where the old-timers meet, exchange ideas, drink a few beers and have a good time—while the computers do all the hard brain work. In some ways, computer chess is much more fun than real chess. You can talk, discuss the game or even ask someone else to play while you go away for an hour. Most professional chess programmers are awful chess players anyway. In fact most of them cannot even beat their own programs!

Notes:

Appendix B
CHESS RULES

Following are the official rules of chess, which were provided by George Koltanowski.

Introduction

The game of chess is a board game played by two opponents who maneuver their chess pieces in an attempt to checkmate the opposing king.

The Chess Board and its Arrangement

1. The chess board is composed of 64 equal squares alternately light (the "white" squares) and dark (the "black" squares).
2. The chess board is placed between the two players in a way so that the square at the right-hand corner to each player is white.
3. The eight rows of squares, running from the edge of the chess board nearest one of the players to the edge of the board nearest the other player, are called *files* (vertical columns).
4. The eight rows of squares, running from one edge of the chess board to the other at right angles to the files, are called *ranks* (horizontal columns).
5. The straight rows of squares of one color, touching each other at the corners, are called *diagonals*.

The Pieces and Their Arrangement

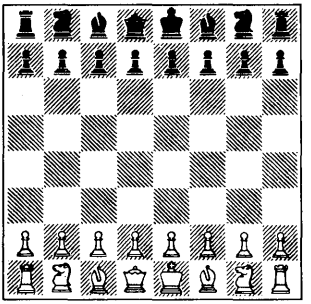
At the start of play, one player uses the 16 light-colored pieces ("white" pieces), and the other player uses the 16 dark-colored pieces ("black" pieces).

These pieces are as follows:

NAMES	WHITE	BLACK
A KING		
A QUEEN		
TWO ROOKS		
TWO BISHOPS		
TWO KNIGHTS		
EIGHT PAWNS		

The initial position of the pieces upon the chess board is as shown in the following diagram:

BLACK



WHITE

The Conduct of the Game

1. The two players must alternate in making one move at a time. The player with the white pieces begins the game.
2. It is said that a player "has the move" when it is his or her turn to play.

The General Definition of the Move

1. Except for castling, a move is the transferring of a piece from one square to another square, vacant or occupied only by an enemy piece. *(An "enemy" piece is a piece of the opposite color.)*
2. No piece, except the rook in castling and the knight can move over a square occupied by another piece.
3. A piece moved to a square occupied by an enemy piece captures this piece, which must be immediately removed from the chess board by the player who makes the capture.

The Individual Moves of the Pieces

The King

Except for castling, the king moves from his square to one of the contiguous squares not under attack by an enemy piece.

A square is "under attack by an enemy piece" when that piece can move to the square on its next move.

Castling is a transfer of the king, completed by the transfer of the rook, counting as a single move (of the king) and executed strictly as follows:

The king moves from his initial square two squares to one side, then the rook from that side passes over the king to occupy the square the king has just passed over.

Castling is definitely not allowed on either side if the king or rook has already moved.

Castling is momentarily prevented:

a) if the initial square of the king or the square which the king must pass over or that which it will occupy is attacked by an enemy piece, or

b) if there are any pieces between the king and the rook toward which the king must move.

The Queen

The queen moves along the length of the rank or file or diagonal upon which she stands.

The Rook

The rook moves along the length of the rank or file upon which it stands.

The Bishop

The bishop moves along the length of the diagonal upon which it stands.

The Knight

The knight moves either two squares on the rank and one on the file, or two on the file and one on the rank (an "L" shape).

The Pawn

The pawn moves as described below:

a) From its initial square, it advances one or two vacant squares forward on its file. Thereafter, it moves only one vacant square on its file at a time. To capture, it advances to a square contiguous to its own upon the diagonal.

b) A pawn attacking a square passed over by an enemy pawn, which has been advanced two squares in one move from its initial square, can capture but only in the move immediately following this enemy pawn, as if that pawn had only moved forward one square. This capture is called "taking in passing" (*"prise en passant"*).

c) Any pawn that reaches the last (eighth) rank must be changed immediately, as a part of the same move, into a queen, rook, bishop or knight of the same color, at the choice of the player and without reference to the other pieces remaining upon the chess board. This changing of a pawn is called "promotion."

Completion of the Move

The completion of a move is achieved:

a) in the transfer of a piece to a vacant square, when the player releases the piece.

b) in a capture, when the captured piece has been removed from the chess board and the player, having placed it on its new square, has released it.

c) in castling, when the player has released the rook upon the square passed over by the king; when the player has released the king, the move is not yet completed, but the player no longer has the right to make any move other than castling.

d) in the promotion of a pawn, when the pawn has been removed from the chess board and the player has released the new piece, placed upon the square of promotion. If the player has released the pawn upon its arrival at the square of promotion, the move is not yet completed, but the player no longer has the right to move a pawn to another square.

The Touched Piece

The player having the move can adjust one or several pieces after warning his or her oponent.

Otherwise, if a player having the move touches one or several pieces, he or she must make a move using the first piece touched which can be moved or captured.

Illegal Positions

1. If a move is made illegally and if one of the players states this fact before touching a piece, the illegality will be corrected applying the rules under the *Touched Pieces* section above. If the illegality is not stated, the game continues without correction.
2. If, in the course of a game, one or several pieces have been accidentally jarred and incorrectly replaced, or if, after an adjournment, the position is incorrectly set-up and if one of the players states this fact before touching a piece, the irregularity can be corrected. If the irregularity is not caught before a player touches a piece, the game continues without correction.
3. If during the game it is claimed that the initial position of the chess board was incorrect, the game will be annulled.

Check

1. The king is in check when his square is attacked by an enemy piece; it is said that this piece gives check to the king.
2. Check must be parried by the move immediately following. If check cannot be parried, it is said to be "mate."
3. A piece interposing to prevent check to the king of its own color can itself give check to the enemy king.

The Won Game

1. The game is won by the player who gives mate to the enemy king.
2. The game will be considered as won by the player whose opponent resigns from the game.

The Drawn Game

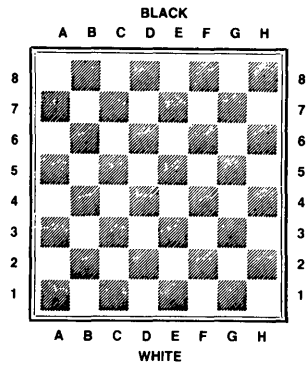
The game is a draw:

1. When the king of the player who has the move is not in check but the player cannot make any move. It is said the king is "stalemated."
2. By agreement between the two players.
3. Upon demand by one of the players when the same position appears three times with the same player having the move each of the three times. The position is considered the same if the pieces of the same denomination and of the same color occupy the same squares.
4. When a player who has the move demonstrates that at least fifty moves have been played by himself and opponent without the capture of any piece or the moving of any pawn.

Systems of Chess Notation

The two most widespread systems of notation are the algebraic system and the descriptive system.

The Algebraic System



The pieces, except the pawns, are designated by their initials. The pawns are not specifically indicated.

(*In American usage knight is indicated by Kt or N, since k indicates the king.*)

The eight files (from the first rank of the white pieces) are designated by the letters a to h.

The eight ranks are numbered from 1 to 8 in counting from the first rank of the white pieces.

(In the original position, the white pieces are found upon the first and second rank, and the black pieces on the seventh and eighth ranks.)

Each square is thus defined by the combination of a letter and a numeral.

To the initial of each piece (except a pawn) one adds the square of departure and the square of arrival. In abbreviated notation, the square of departure is omitted.

Thus Bc1-f4 means the bishop upon the square c1 is moved to the square f4. In abridged notation: Bf4.

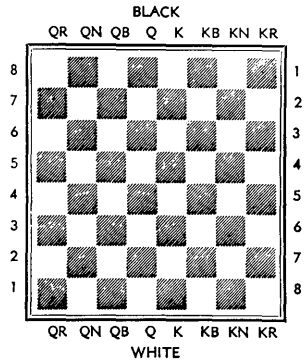
In another example, e7 – e5 means the pawn upon the square e7 is moved to the square e5. In abridged notation: e5.

When two pieces of the same denomination can be moved to the same square, the abridged notation is expanded in the following manner: If, for example, two knights are at g1 and d2, the move Ktg1 – f3 is written in abridged as Ktg – f3. If the knights were at g1 and g5, the move Ktg1 – f3 would be abridged as Kt1 – f3.

ABBREVIATIONS

O – O	Castling with the rook on h1 or h8 (short castling)
O – O – O	Castling with the rook on a1 or a8 (long castling)
: or x	Capturing
+	Giving check
+ +	Giving checkmate
!	Well played
?	Poorly played

The Descriptive System



As in the algebraic system, the pieces, except the pawns, are designated by their initials. The pawns are not specially indicated.

(*In American usage and notation the pawns are also indicated by their initial; and the knight is indicated by Kt or N, since K is reserved to designate the king.*)

One distinguishes the rook, knight and bishop of the king and the queen by the addition of the letters K and Q (KR or QR, etc.).

The eight files (from left to right for the white pieces and right to left for the black pieces) are distinguished thus:

File of the queen's rook (QR)

File of the queen's knight (Qkt or QN)

File of the queen's bishop (QB)

File of the queen (Q)

File of the king (K)

File of the king's bishop (KB)

File of the king's knight (KKt or KN)

File of the king's rook (KR)

The eight ranks are numbered from 1 to 8, in counting from the first rank, for the white and black pieces.

The initial of the piece moved and the square to which it moves are indicated.

Example: Q-KB4 means the queen is moved to the fourth square in the file of the king's bishop.

When two pieces of the same denomination can move to the same square, the square of departure and the square of arrival are both indicated. Thus R(KKt4)-KKt2 means the one of the two rooks which is on the fourth square of the king's knight's file is moved to the second square of the same file.

(*In practice, the indication in abridged form of R(Kt) or R(4) is usually sufficient, if the two rooks are not both on the Kt file or both on the fourth rank.*)

ABBREVIATIONS

O – O	Castling with the king's rook (short castling)
O – O – O	Castling with the queen's rook (long castling)
: or x	Capturing
ch. or +	Giving check
!	Well played
?	Poorly played

Notes:

Appendix C
BASIC BRIDGE RULES AND STRATEGY

Following are the official rules of bridge, provided by Kit Woolsey.

Bridge is a card game for four players, two pairs of partners. The partners sit across the table from one another, with an opponent on each side. For convenience, we will refer to the players as "North," "East," "South," and "West" to correspond with their geographical positions around the table. Thus North and South are partners, East and West are partners, and around the table clockwise are North, East, South, and West.

Each player is dealt 13 cards. The object of the game is for one partnership to contract to win a certain number of *tricks*, while the opposing pair attempts to prevent this. A trick consists of four cards, one played by each player. Since each player has 13 cards, there are 13 tricks available.

The play to a trick is as follows: one player plays any card, and the other three players each play a card in turn, going clockwise around the table. A player must "follow suit," that is, play a card of the same suit as the first card of the trick if s/he is able to do so—if not, s/he may play any card. The winner of the trick is the player who plays the highest card of the suit led (ace is high, deuce is low), unless there is a *trump suit* (a special suit determined by the bidding) in which case the highest trump (if any) played wins the trick. The person who wins a trick plays first to the next trick, and play continues until all 13 tricks have been played.

The determination of the trump suit and which pair contracts for how many tricks is done by an auction—the highest bidder gets the contract. The rank order of the suits is clubs (lowest), then diamonds, hearts, spades and notrump (highest). *Notrump* means what it sounds like—no trump suit. The smallest number of tricks one can contract for is seven. Six is added to the bid, and that is the number of tricks which the bidder is contracting to win. For example, "two spades" contracts to take eight tricks with spades as the trump suit; "three notrump" contracts to take nine tricks with no trump suit.

If two bids contract for the same number of tricks, the one with the higher ranking suit is the higher bid (e.g., three hearts is higher than three diamonds), but a bid for more tricks always outranks a bid for fewer tricks (e.g., four clubs outranks three spades). You must contract for at least seven tricks if you bid, so the lowest possible bid is

one club, and since there are only thirteen tricks available, the highest possible bid is seven notrump. If you bid, you must always make a higher ranking bid than the previous bid.

The auction proceeds as follows: starting with the dealer, and going clockwise, each player has the option of either passing or making a bid higher than the last bid. In addition, if an opponent has made the last bid you have the option of *doubling*, and if an opponent has doubled your bid you have the option of *redoubling*. These doubles and redoubles increase both the rewards for making the contract and the penalties for defeating the contract. The bidding continues until there are three consecutive passes, unless the first three bids are passes—in which case the fourth player has a chance to bid. If he also passes, the hand is redealt. You are permitted to make a legitimate bid even if you have passed earlier in the auction.

When the bidding has concluded, the pair which made the highest bid is the *declaring side*, and the player of that pair who first named the suit of the final contract is the *declarer*. For example, if the auction goes:

North	**East**	**South**	**West**
One spade	Pass	Two hearts	Pass
Three hearts	Pass	Pass	Pass

then South is the declarer at three hearts, even though North made the final bid—South bid hearts first. To make the contract, South must take at least nine tricks with hearts as the trump suit.

The play proceeds as follows: the player to the left of the declarer leads to the first trick. After this lead, declarer's partner (called the *dummy*) puts his or her cards face up on the table. The dummy has no further say in the proceedings—the declarer plays both dummy's cards and his or her own. Play to the first trick continues with declarer playing a card from dummy, then the player to declarer's right playing to the trick, and finally declarer playing the trick. The winner of the trick leads to the next trick, and play continues until all 13 tricks have been completed. If declarer has won at least the number of tricks contracted for, s/he has made the contract; if not, s/he has been defeated.

If the declaring side fulfills their contract, they score points, determined by the number of tricks taken, the contract, and the trump suit. Only tricks taken over the sixth trick are counted. The lower suits (clubs and diamonds) are called *minor suits*, and the higher suits (hearts and spades) are called *major suits*. Each trick (over six) in a minor suit is worth 20 points, and each trick in a major suit is worth 30 points. In notrump, the first trick is worth 40 points, and all other tricks are worth 30 points.

There are two kinds of scores. Tricks that have been contracted for are scored "*below the line*"; extra tricks and other bonuses are all "*above the line.*" For example, suppose the North-South pair bid two notrump and win ten tricks. Their score is 70 points below the line (40 for the first trick above six, 30 for the next) and 60 points above the line, since they only contracted for eight tricks.

If a pair scores 100 or more points below the line, they have won a game. The first pair to win two games wins the *rubber* (two out of three games), and scores a bonus of 700 points if their opponents have not won a game that rubber, 500 points if their opponents have won a game. This is a large bonus, so it is important to contract for enough tricks to win a game (called *bidding a game*) if you think you can take them. You need 11 tricks for game in a minor suit, 10 tricks for game in a major suit and 9 tricks for game in notrump. If you contract for fewer tricks (called a *part-score*) and make the contract, your score below the line carries over to the next deal, but once either pair wins a game both pairs must start from scratch for the 100 points.

If the contract is defeated, the defenders score points. If the declaring side has not scored a game in the rubber (they are non-vulnerable) then the penalty is 50 points for each trick the contract failed by, while if they have scored a game (are vulnerable) the penalty is 100 points for each trick.

There is an additional bonus for contracting and making 12 tricks (a small slam) or all 13 tricks (a grand slam). The small slam bonus is 500 points if non-vulnerable, 750 points if vulnerable. The grand slam bonus is 1000 points if non-vulnerable, 1500 points if vulnerable.

Doubles and redoubles increase the scores and penalties. If the contract is made, each trick that is contracted for scores double, while each extra trick scores 100 points if non-vulnerable, and 200 points if vulnerable. In addition, there is a 50 point bonus for making a doubled or redoubled contract. If the contract is defeated, the defenders score 100 points for defeating the contract one trick, and 200 points for each subsequent undertrick if declarer is non-vulnerable; 200 points for the first undertrick and 300 points for each subsequent undertrick if vulnerable. Redoubles double all the doubled scores, except the 50 point bonus for making the contract remains the same.

The primary goal in the bidding is to choose the final contract such that the partnership's best suit is trumps, and to contract for game if the partnership can win enough tricks. Since high cards win tricks, evaluation of the strength of a hand depends primarily upon the high card content. The evaluation technique used by almost all players today is as follows:

ace	=	4
king	=	3
queen	=	2
jack	=	1

Notice that there are a total of 40 high card points, so an average hand contains 10 high card points. You usually want a hand to be at least one king above average, 13 or more points, to make the first bid, although this requirement can be relaxed when you have a long suit (5 or 6 cards), since long suits produce extra tricks in the play. Experience has shown that a partnership's combined total of 26 points is usually sufficient to make a game in notrump (9 tricks) or in a major suit (10 tricks), but a minor suit game (11 tricks) usually needs 29 points. Consequently, if your partner opens the bidding and you have 13 or more points you should be sure to arrive at some game contract. A trump suit will be worth an extra trick or two in the play if you and your partner have at least eight trumps between the two hands. If no satisfactory trump suit can be found, then you should play in notrump.

When declaring a contract, it is important to plan your line of play when the dummy's hand is placed on the table. Do *not* play a card from dummy until you know what you are going to do for the rest of the hand. Doing so is the single most common mistake in bridge.

In a trump contract, determine the number of tricks you are likely to lose in each suit, and if this is more than you can afford to lose, attempt to avoid these losers, perhaps by trumping them in the dummy or discarding them on some of dummy's winners. It will usually be correct strategy to first lead trumps until the opponents have no more trumps—you don't want them trumping your aces and kings. Remember to take advantage of the clockwise order of play. For example, if you hold the king of a suit in your hand and your right hand opponent holds the ace, you will not win a trick with the king if you lead it, but if you lead the suit from dummy you can now play small if s/he plays the ace and play the king if s/he doesn't.

If you are declaring a notrump contract, you should count the number of tricks you expect to win in each suit. If the total is not enough to make your contract, you must establish some more winners. This is usually done by playing the suit in which your hand and the dummy have the greatest combined length. Since there is no trump suit, a lowly deuce will win the trick if nobody has any more of the suit, so attack your long suits. Knock out your opponent's aces and kings, and your smaller cards will score tricks. Remember that the opponents will be trying to do the same thing. Notrump contracts are often a race to see which side can set up their long suit first.

Appendix D
SUGGESTED READING

Bridge

Ewen, Robert B. *Charles H. Goren Presents the Precision System of Contract Bridge*. Doubleday, 1971.

Goren, Charles Henry. *Goren's Bridge Complete*. Doubleday, 1971.

Goren, Charles Henry. *New Contract Bridge in a Nutshell*. Doubleday, 1972.

Goren, Charles Henry. *Play as You Learn Bridge*. Doubleday, 1979.

Goren, Charles Henry and Olson, Jack. *Bridge is my Game: Lessons of a Lifetime*. Doubleday, 1965.

Lawrence, Mike. *How to Read Your Opponent's Card: The Bridge Expert's Way to Locate Missing High Cards*. Prentice-Hall, 1973.

Morehead, Albert Hodges. *Morehead on Bidding*. (Morehead's classic book revised and modernized by Richard L. Frey.) Simon and Schuster, 1974.

Chess

Frey, Peter W., ed. *Chess Skill in Man and Machine*, 2nd edition. Springer-Verlag, 1983.

Harkness, Kenneth. *Official Chess Rulebook*. McKay, 1970.

Levy, David. *Chess and Computers*. Computer Science Press, 1976.

Reinfeld, Fred and Irving Chernev. *Chess Strategy and Tactics*. David McKay Company, Inc., 1946.

Sunnucks, Anne, ed. *The Encyclopedia of Chess*. St. Martin's Press, 1976.

Go-Moku

Lasker. Edward. *Go and Go-Moku, The Oriental Board Games*. Dover Publications, 1960.

Appendix E
GLOSSARY

Above the line: A bridge term referring to the scoring of extra tricks and other bonuses.

Algebraic notation: Chess notation using ranks numbered from 1 to 8 and files labeled from *a* to *h*. A particular square on the board can then be indicated as, e.g., *D6*. This is different from common chess notation; knight to king's bishop 3 would be noted algebraically as *NKB3*.

Algorithm: Method of solving a particular problem; in programming, the plan for performing a calculation that the programming code itself will perform during program execution.

Alpha-Beta search: A search algorithm designed to arrive at the same conclusion as the Minimax search, but without having to evaluate all branches of the decision tree. See *Minimax* search.

Alpha-Beta window: Rather than using infinity (or, more practically, negative and positive *MaxInt*) for *alpha* and *beta*, the Alpha-Beta window attempts to cut off less productive branches of a decision tree from the search by narrowing the limits of the values applied to *alpha* and *beta*.

Array: A list of like elements indexed by a value or group of values. Arrays have one, two or more dimensions.

Attack value: In Turbo Chess, a value signifying the ability of a piece to attack another, used to calculate positional superiority of one side to the other.

Below the line: A bridge term referring to the scoring of tricks that have been contracted for.

Best line: In Turbo Chess, the series of moves the program expects the game to follow.

Bid: In bridge, an offer to take a specific number of tricks of a certain number or suit.

Bidding a game: A bridge term meaning to contract for enough tricks to win a game.

Blackwood: In bridge, a method of asking your partner how many aces or kings s/he holds. Used for slam bidding only. See *Slam*.

Blitz chess: Chess played with only seconds allowed to make a move, and no time to create deep plans and strategies (from WWII *Blitzkrieg*, lightning war).

Brute Force: Any process conducted on a computer with no regard for saving time, effort or minimizing steps; exhaustive and thorough, but not elegant.

Capture search: A search conducted in Turbo Chess to determine if there are any pieces to capture.

Castling: In chess, a move in which the king and rook approximate changing places in a single move. Used to place the king in a more protected position.

Contract: In bridge, the final bid.

Cut off: In a tree search, a cut off prevents the search from traveling down less productive branches.

Data structure: The organization of constants, types, and variables to best represent the data with which the programmer is working.

Declarer: In bridge, the first of the pair winning the contract to bid the contract suit.

Distribution: In bridge, the pattern or shape formed by the number of cards held—for example, 5-5-1-2 or 2-6-3-2.

Doubleton: In bridge, when only two cards of a specific suit are held.

Dummy: In bridge, partner of the declarer.

ELO points: Scale used to measure the relative playing strength of chess players. A factor of two in speed translates to about 60 ELO points, while a factor of ten in speed equals about 200 ELO points. A difference of 100 ELO points between players means the better player should win 2/3 of the time.

Endgame: In chess, the last part of the game, with few pieces remaining on the board.

Endless loop: See *loop, endless.*

En passant capture: "In passing." In chess, a move in which one pawn passes an opposing pawn and captures it by moving to the square behind.

En prise: "Taken."

Evaluator: The procedure used by a game program to determine which of several potential moves is the best.

Evaluation spice: Code added to the evaluation algorithm to handle special situations.

Fifty move rule: In chess, a draw game resulting from the player who has the move demonstrating that at least 50 moves have been played by him/herself and the opponent without the capture of any piece or the moving of any pawn.

Files: The vertical columns of a chess or checkerboard.

Follow suit: In bridge, playing a card of the same suit as the first card of the trick.

Heuristic: Problem-solving technique that uses rules of strategy; used by Turbo GameWorks to guide the tree search.

Hint: What a game program may give you when it applies its computerized evaluation capability to your situation rather than its own.

Horizon effect: An effect caused when a game evaluation procedure selects a supposedly sound move at the limit of its "lookahead" (say four plies), but on the move following that, the opponent captures the piece. (In this case, the four-ply "horizon" causes a less than ideal evaluation).

Incremental updating: Intelligently updating only the part of a data structure representation that a move affects.

Initialize: Provide the initial values for variables at the beginning of a procedure or program.

Killer moves: Moves that the evaluation procedure has discovered an opponent can use against the program. The next time the program evaluates moves, the killer moves are checked first to make sure that the program has them blocked. Eliminates some search time.

Loop, endless: See *endless* loop.

Major suit: In bridge, spades or hearts.

Material: The number of your pieces still on the board versus those the opponent has captured.

Midgame: In chess, the longest period of play (between the opening and the endgame).

Minimax search: Search algorithm that finds the best move, assuming that the opponent will play his or her best possible move.

Minor suits: In bridge, clubs and diamonds.

Move generator: A procedure that produces all the legitimate moves possible for all the (program's) pieces on the board.

Nesting: Placing one loop or procedure inside another. For example, in:

```
for i := 1 to 100 do
   begin
     write(i);
     for n := 1 to 5 do
       write('Hello');
   end;
```

The *n* loop is nested within the *i* loop, and executes five times for each increment of *i*.

Node: In a tree search, the point where multiple possible moves diverge.

Non-vulnerable: In bridge, the side that has not yet scored a game in the current rubber.

Normal move: As used by Turbo Chess, any move that is not a special move. See *special move*.

Notrump: In bridge, no trump suit.

Opening library: A list of opening chess moves. At the beginning of a chess game, the many possible moves with very minor differences in value can make searches lengthy. Turbo Chess solves this by using a list of pre-programmed openings and replies.

Overtrick: In bridge, a trick that exceeds the contract.

Part score: In bridge, contracting for fewer tricks than are required to win a game.

Pawn structure: The arrangement of pawns on a chessboard as it relates to their ability to guard and capture.

Ply: One half-move (the move of one person in a two-person game). A full move is when both players have had a turn.

Positional play: A game strategy dependent on the placement of pieces on a playing board and the relative value of those pieces.

Ranks: The horizontal rows of a chess or checkerboard.

Rubber: In bridge, a series of hands that ends when either side scores two games.

Ruff: In bridge, to play a trump on a side when you cannot follow suit.

Search tree: In a two-person game, a "map" of the possible moves open to one player and the replies for each of those moves available to the other player. Each level of move is called a ply.

Shannon B: An algorithm that restricts the number of moves searched at each node in order to conduct a deeper search.

Side suit: In bridge, any suit except trump.

Singleton: In bridge, when only one card of a specific suit is held.

Slam: In bridge, a contract for twelve or thirteen tricks.

Spaghetti code: A computer program so convoluted that trying to follow the flow of logic is like tracing one strand in a bowl of spaghetti.

Stayman convention: A special bid in bridge (two clubs in response to one NT or three clubs in response to two NT) that shows nothing about the club suit, but asks the partner to show four-card suits in hearts or spades.

Suit: In bridge, the thirteen cards that all have the same symbol (either clubs, diamonds, spades or hearts).

Third repetition: In chess, a game drawn upon demand by one of the players when the same position appears three times with the same player having the move each of the three times. The position is considered the same if the pieces of the same denomination and of the same color occupy the same squares.

Tolerance search: The smaller the difference in value between two good moves, the longer it takes a program to decide between them. To avoid this situation, the program is satisfied if it finds one of the best moves, rather than the absolute best move.

Top down programming: A program design practice of starting from the general and working towards the specific.

Trick: In bridge, four cards played in a single round, one played by each player.

Trump suit: In bridge, a special suit determined by the bidding that outranks all other suits for the duration of a hand.

Void suit: In bridge, when no cards of a certain suit are held.

Vulnerable: A bridge term referring to the side that has scored one game towards the current rubber.

SUBJECT INDEX

U

60 DAY MONEY-BACK GUARANTEE

CATALOG OF BORLAND PRODUCTS

BORLAND
INTERNATIONAL
4585 Scotts Valley Drive
Scotts Valley, CA 95066

Available at better dealers nationwide. Call (800) 556-2283 for the dealer nearest you. To order by Credit Card call (800) 255-8008, CA (800) 742-1133

SIDEKICK® VERSION 1.5

INFOWORLD'S SOFTWARE PRODUCT OF THE YEAR

Whether you're running WordStar™, Lotus™, dBase™, or any other program, SIDEKICK puts all these desktop accessories at your fingertips. Instantly.

A full-screen WordStar-like Editor You may jot down notes and edit files up to 25 pages long.

A Phone Directory for your names, addresses and telephone numbers. Finding a name or a number becomes a snap.

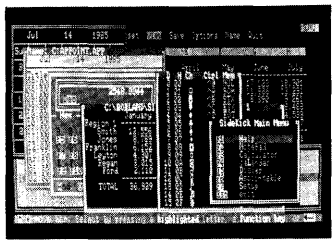
An Autodialer for all your phone calls. It will look up and dial telephone numbers for you. (A modem is required to use this function.)

A Monthly Calendar functional from year 1901 through year 2099.

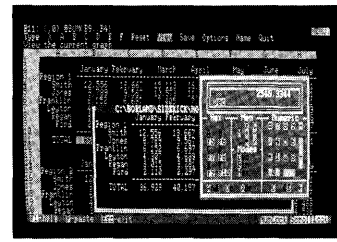
A Datebook to remind you of important meetings and appointments.

A full-featured Calculator ideal for business use. It also performs decimal to hexadecimal to binary conversions.

An ASCII Table for easy reference.



All the SIDEKICK windows stacked up over Lotus 1-2-3. From bottom to top: SIDEKICK'S "Menu Window," ASCII Table, Notepad, Calculator, Datebook, Monthly Calendar and Phone Dialer.



Here's SIDEKICK running over Lotus 1-2-3. In the SIDEKICK Notepad you'll notice data that's been imported directly from the Lotus screen. In the upper right you can see the Calculator.

The Critics' Choice

"In a simple, beautiful implementation of WordStar's™ block copy commands, SIDEKICK can transport all or any part of the display screen (even an area overlaid by the notepad display) to the notepad."
—Charles Petzold, PC MAGAZINE

"SIDEKICK deserves a place in every PC."
—Garry Ray, PC WEEK

"SIDEKICK is by far the best we've seen. It is also the least expensive." **—Ron Mansfield, ENTREPRENEUR**

"If you use a PC, get SIDEKICK. You'll soon become dependent on it." **—Jerry Pournelle, BYTE**

SIDEKICK IS AN UNPARALLELED BARGAIN AT ONLY $54.95 (copy-protected) OR $84.95 (not copy-protected)

Minimum System Configuration: SIDEKICK is available now for your IBM PC, XT, AT, PCjr., and 100% compatible microcomputers. The IBM PC jr. will only accept the SIDEKICK not copy-protected version. Your computer must have at least 128K RAM, one disk drive and PC-DOS 2.0 or greater. A Hayes™ compatible modem, IBM PCjr.™ internal modem, or AT&T® Modem 4000 is required for the autodialer function.

BORLAND INTERNATIONAL

SuperKey®

INCREASE YOUR PRODUCTIVITY BY 50% OR YOUR MONEY BACK

SuperKey turns 1,000 keystrokes into 1!

Yes, SuperKey can *record* lengthy keystroke sequences and play them back at the touch of a single key. Instantly. Like Magic.

Say, for example, you want to add a column of figures in 1-2-3. Without SuperKey you'd have to type seven keystrokes just to get started. ["shift-@-s-u-m-shift-("]. With SuperKey you can turn those 7 keystrokes into 1.

SuperKey keeps your 'confidential' files. . .CONFIDENTIAL!

Time after time you've experienced it: anyone can walk up to your PC, and read your confidential files (tax returns, business plans, customer lists, personal letters. . .).

With SuperKey you can encrypt any file, even while running another program. As long as you keep the password secret, only YOU can decode your file. SuperKey implements the U.S. government Data Encryption Standard (DES).

SuperKey helps protect your capital investment.

SuperKey, at your convenience, will make your screen go blank after a predetermined time of screen/keyboard inactivity. You've paid hard-earned money for your PC. SuperKey will protect your monitor's precious phosphor. . .and your investment.

SuperKey protects your work from intruders while you take a break.

Now you can lock your keyboard at any time. Prevent anyone from changing hours of work. Type in your secret password and everything comes back to life. . .just as you left it.

SUPERKEY is now available for an unbelievable $69.95 (not copy-protected).

Minimum System Configuration: SUPERKEY is compatible with your IBM PC, XT, AT, PCjr. and 100% compatible microcomputers. Your computer must have at least 128K RAM, one disk drive and PC-DOS 2.0 or greater.

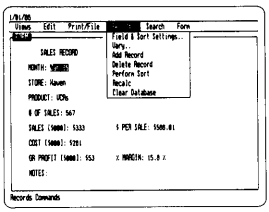
BORLAND
INTERNATIONAL

REFLEX

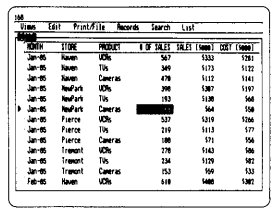
THE ANALYST™

Reflex™ is the most amazing and easy to use database management system. And if you already use Lotus 1-2-3, dBASE or PFS File, you need Reflex—because it's a totally new way to look at your data. It shows you patterns and interrelationships you didn't know were there, because they were hidden in data and numbers. It's also the greatest report generator for 1-2-3.

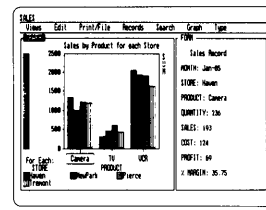
REFLEX OPENS MULTIPLE WINDOWS WITH NEW VIEWS AND GRAPHIC INSIGHTS INTO YOUR DATA.



The FORM VIEW lets you build and view your database.

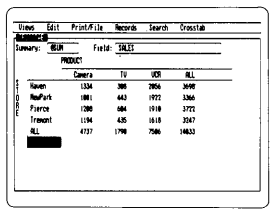


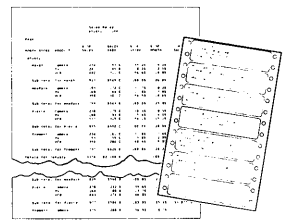
The LIST VIEW lets you put data in tabular List form just like a spreadsheet.



The GRAPH VIEW gives you instant interactive graphic representations.

The CROSSTAB VIEW gives you amazing "cross-referenced" pictures of the links and relationships hidden in your data.





The REPORT VIEW allows you to import and export to and from Reflex, 1-2-3, dBASE, PFS File and other applications and prints out information in the formats you want.

So Reflex shows you. Instant answers. Instant pictures. Instant analysis. Instant understanding.

THE CRITICS' CHOICE:

"The next generation of software has officially arrived."
Peter Norton, PC WEEK

"Reflex is one of the most powerful database programs on the market. Its multiple views, interactive windows and graphics, great report writer, pull-down menus and cross tabulation make this one of the best programs we have seen in a long time . . .

The program is easy to use and not intimidating to the novice . . . Reflex not only handles the usual database functions such as sorting and searching, but also "what-if" and statistical analysis . . . it can create interactive graphics with the graphics module. The separate report module is one of the best we've ever seen."
Marc Stern, INFOWORLD

Minimum System Requirements: Reflex runs on the IBM® PC, XT, AT and compatibles. 384K RAM minimum. IBM Color Graphics Adapter®, Hercules Monochrome Graphics Card™, or equivalent. PC-DOS 2.0 or greater. Hard disk and mouse optional. Lotus 1-2-3, dBASE, or PFS File optional.

BORLAND
INTERNATIONAL

If you use an IBM PC, you need

TURBO Lightning™

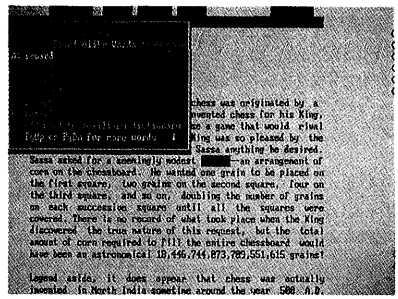
Turbo Lightning™ teams up with the Random House spelling Dictionary® to check your spelling as you type!

Turbo Lightning, using the 83,000-word Random House Dictionary, checks your spelling *as you type*. If you misspell a word, it alerts you with a 'beep'. At the touch of a key, *Turbo* Lightning opens a window on top of your application program and suggests the correct spelling. Just press ENTER and the misspelled word is instantly replaced with the correct word. It's that easy!

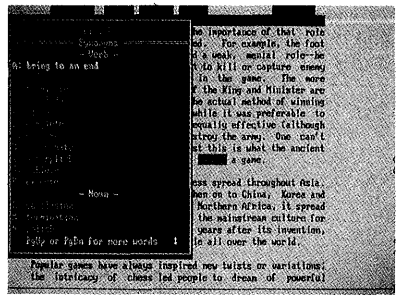
Turbo Lightning works hand-in-hand with the Random House Thesaurus® to give you instant access to synonyms.

Turbo Lightning lets you choose just the right word from a list of alternates, so you don't say the same thing the same way every time. Once *Turbo* Lightning opens the Thesaurus window, you see a list of alternate words, organized by parts of speech. You just select the word you want, press ENTER and your new word will instantly replace the original word. Pure magic!

If you ever write a word, think a word, or say a word, you need Turbo Lightning.



The Turbo Lightning Dictionary.



The Turbo Lightning Thesaurus.

Turbo Lightning's intelligence lets you teach it new words. The more you use Turbo Lightning, the smarter it gets!

You can also *teach* your new *Turbo* Lightning your name, business associates' names, street names, addresses, correct capitalizations, and any specialized words you use frequently. Teach *Turbo Lightning* once, and it knows forever.

Turbo Lightning™ is the engine that powers Borland's Turbo Lightning Library™.

Turbo Lightning brings electronic power to the Random House Dictionary® and Random House Thesaurus®. They're at your fingertips —even while you're running other programs. *Turbo* Lightning will also 'drive' soon-to-be-released encyclopedias, extended thesauruses, specialized dictionaries, and many other popular reference works. You get a head start with this first volume in the *Turbo* Lightning Library.

And because *Turbo* Lightning is a Borland product, you know you can rely on our quality, our 60-day money-back guarantee, and our eminently fair prices.

BORLAND INTERNATIONAL

Suggested Retail Price $99.95
(not copy-protected)

Minimum System Requirements:
128K IBM PC® or 100% compatible computer, with 2 floppy disk drives and PC-DOS (MS-DOS) 2.0 or greater.

SideKick, the Macintosh Office Manager, brings information management, desktop organization and telecommunications to your Macintosh. Instantly, while running any other program.

A full-screen editor/mini-word processor lets you jot down notes and create or edit files. Your files can also be used by your favorite word processing program like MacWrite™ or MicroSoft® Word™.

A complete telecommunication program sends or receives information from any on-line network or electronic bulletin board while using any of your favorite application programs. A modem is required to use this feature.

A full-featured financial and scientific calculator sends a paper-tape output to your screen or printer and comes complete with function keys for financial modeling purposes.

A print spooler prints *any* text file while you run other programs.

A versatile calendar lets you view your appointments for a day, a week or an entire month. You can easily print out your schedule for quick reference.

A convenient "Things-to-Do" file reminds you of important tasks.

A convenient alarm system alerts you to daily engagements.

A phone log keeps a complete record of all your telephone activities. It even computes the cost of every call. Area code hook-up provides instant access to the state, region and time zone for all area codes.

An expense account file records your business and travel expenses.

A credit card file keeps track of your credit card balances and credit limits.

A report generator prints-out your mailing list labels, phone directory and weekly calendar in convenient sizes.

A convenient analog clock with a sweeping second-hand can be displayed anywhere on your screen.

On-line help is available for all of the powerful SIDEKICK features.

Best of all, everything runs concurrently.

SIDEKICK, the software Macintosh owners have been waiting for.

SideKick, Macintosh's Office Manager is available now for $84.95 (not copy-protected).

Minimum System Configuration: SIDEKICK is available now for your Macintosh microcomputer in a format that is not copy-protected. Your computer must have at least 128K RAM and one disk drive. Two disk drives are recommended if you wish to use other application programs. A Hayes-compatible modem is required for the telecommunications function. To use SIDEKICK'S autodialing capability you need the Borland phone-link interface. See inside for details.

FREE MICROCALC SPREADSHEET
WITH COMMENTED SOURCE CODE !

TURBO PASCAL®

VERSION 3.0

THE CRITICS' CHOICE:

"Language deal of the century . . . Turbo Pascal: it introduces a new programming environment and runs like magic."
—Jeff Duntemann, PC Magazine

"Most Pascal compilers barely fit on a disk, but Turbo Pascal packs an editor, compiler, linker, and run-time library into just 39K bytes of random-access memory."
—Dave Garland, Popular Computing

"What I think the computer industry is headed for: well - documented, standard, plenty of good features, and a reasonable price." ***—Jerry Pournelle, BYTE***

THE FEATURES:

One-Step Compile: No hunting & fishing expeditions! Turbo finds the errors, takes you to them, lets you correct, then instantly recompiles. You're off and running in record time.

Built-in Interactive Editor: WordStar-like easy editing lets you debug quickly.

Automatic Overlays: Fits big programs into small amounts of memory.

Microcalc: A sample spreadsheet on your disk with ready-to-compile source code.

IBM PC VERSION: Supports Turtle Graphics, Color, Sound, Full Tree Directories, Window Routines, Input/Output Redirection and much more.

LOOK AT TURBO NOW!

- ☐ More than 400,000 users worldwide.
- ☐ TURBO PASCAL is proclaimed as the de facto industry standard.
- ☐ TURBO PASCAL PC MAGAZINE'S award for technical excellence.
- ☐ TURBO PASCAL named 'Most Significant Product of the Year' by PC WEEK.
- ☐ TURBO PASCAL 3.0 — the FASTEST Pascal development environment on the planet, PERIOD.

OPTIONS FOR 16-BIT SYSTEMS:

8087 math co-processor support for intensive calculations.

Binary Coded Decimals (BCD): Eliminates round-off error! A *must* for any serious business application. (No additional hardware required.)

Turbo Pascal 3.0 is available now for $69.95.

Options: Turbo Pascal with 8087 or BCD at a low $109.90. Turbo Pascal with both options (8087 and BCD) priced at $124.95.

MINIMUM SYSTEM CONFIGURATION: To use Turbo Pascal 3.0 requires 64K RAM, one disk drive, Z-80, 8088/86, 80186 or 80286 microprocessor running either CP/M-80 2.2 or greater, CP/M-86 1.1 or greater, MS-DOS 2.0 or greater or PC-DOS 2.0 greater, MS-DOS 2.0 or greater or PC-DOS 2.0 or greater. A XENIX version of Turbo Pascal will soon be announced, and before the end of the year, Turbo Pascal will be running on most 68000-based microcomputers.

BORLAND
INTERNATIONAL

LEARN PASCAL FROM THE FOLKS WHO INVENTED TURBO PASCAL® AND TURBO DATABASE TOOLBOX®.

Borland International proudly introduces **Turbo Tutor®**. The perfect complement to your **Turbo Pascal** compiler. **Turbo Tutor** is *really* for everyone—even if you've never programmed before.

And if you're already proficient, **Turbo Tutor** can sharpen up the fine points. The 300 page manual and program disk divides your study of Pascal into three learning modules:

FOR THE NOVICE: Gives you a concise history of Pascal, tells you how to write a simple program, and defines the basic programming terms you need to know.

ADVANCED CONCEPTS: If you're an expert, you'll love the sections detailing subjects such as "how to use assembly language routines with your **Turbo Pascal** programs."

PROGRAMMER'S GUIDE: The heart of **Turbo Pascal.** This section covers the fine points of every aspect of **Turbo Pascal** programming: program structure, data types, control structures, procedures and functions, scalar types, arrays, strings, pointers, sets, files and records.

A MUST. You'll find the source code for all the examples in the book on the **accompanying disk** ready to compile.

Turbo Tutor may be the only reference on Pascal and programming you'll ever need!

TURBO TUTOR—A REAL EDUCATION FOR ONLY $34.95.

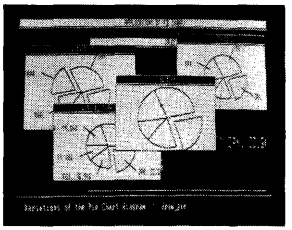
(not copy-protected)

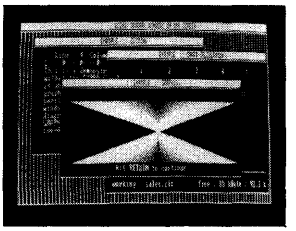
***Minimum System Configuration: TURBO TUTOR is available today for your computer running TURBO PASCAL for PC-DOS, MS-DOS, CP/M-80, and CP/M-86. Your computer must have at least 128K RAM, one disk drive and PC-DOS 1.0 or greater, MS-DOS 1.0 or greater, CP/M-80 2.2 or greater, or CP/M-86 1.1 or greater.**

BORLAND
INTERNATIONAL

TURBO *GRAPHIX* TOOLBOX™

HIGH RESOLUTION GRAPHICS AND GRAPHIC WINDOW MANAGEMENT FOR THE IBM PC





Dazzling graphics and painless windows.

The Turbo Graphix Toolbox™ will give even a beginning programmer the expert's edge. It's a complete library of Pascal procedures that include:

- Full graphics window management.
- Tools that allow you to draw and hatch pie charts, bar charts, circles, rectangles and a full range of geometric shapes.
- Procedures that save and restore graphic images to and from disk.
- Functions that allow you to precisely plot curves.
- Tools that allow you to create animation or solve those difficult curve fitting problems.

No sweat and no royalties.

You can incorporate part, or all of these tools in your programs, and yet, we won't charge you any royalties. Best of all, these functions and procedures come complete with source code on disk ready to compile!

John Markoff & Paul Freiberger, syndicated columnists:

"While most people only talk about low-cost personal computer software, Borland has been doing something about it. And Borland provides good technical support as part of the price."

Turbo Graphix Toolbox—only $54.95 (not copy protected).

Minimum System Configuration: Turbo Graphix Toolbox is available today for your computer running Turbo Pascal 2.0 or greater for PC-DOS, or truly compatible MS-DOS. Your computer must have at least 128K RAM, one disk drive and PC-DOS 2.0 or greater, and MS-DOS 2.0 or greater with IBM Graphics Adapter or Enhanced Graphics Adapter, IBM-compatible Graphics Adapter, or Hercules Graphics Card.

BORLAND
INTERNATIONAL

TURBO DATABASE TOOLBOX™

Is The Perfect Complement To Turbo Pascal.

It contains a complete library of Pascal procedures that allows you to sort and search your data and build powerful applications. It's another set of tools from Borland that will give even the beginning programmer the expert's edge.

THE TOOLS YOU NEED!

TURBOACCESS Files Using B+Trees - The best way to organize and search your data. Makes it possible to access records in a file using key words instead of numbers. Now available with complete source code on disk ready to be included in your programs.

TURBOSORT – The fastest way to sort data—and TURBOSORT is the method preferred by knowledgeable professionals. Includes source code.

GINST (General Installation Program) – Gets your programs up and running on other terminals. This feature alone will save hours of work and research. Adds tremendous value to all your programs.

GET STARTED RIGHT AWAY: FREE DATABASE!

Included on every Toolbox disk is the source code to a working database which demonstrates the power and simplicity of our Turbo Access search system. Modify it to suit your individual needs or just compile it and run. Remember, no royalties!

THE CRITICS' CHOICE!

"The tools include a B+tree search and a sorting system. I've seen stuff like this, but not as well thought out, sell for hundreds of dollars."

—Jerry Pournelle, BYTE MAGAZINE

"The Turbo Database Toolbox is solid enough and useful enough to come recommended."

—Jeff Duntemann, PC TECH JOURNAL

TURBO DATABASE TOOLBOX—ONLY $54.95 (not copy-protected).

Minimum system configurations: 64K RAM and one disk drive. 16-bit systems: TURBO PASCAL 2.0 or greater for MS-DOS or PC-DOS 2.0 or greater. TURBO PASCAL 2.1 or greater for CP/M-86 1.1 or greater. Eight-bit systems: TURBO PASCAL 2.0 or greater for CP/M-80 2.2 or greater.

TURBO EDITOR TOOLBOX

It's All You Need To Build Your Own Text Editor Or Word Processor.

Build your own lightning-fast editor and incorporate it into your Turbo Pascal programs. Turbo Editor Toolbox™ gives you easy-to-install modules. Now you can integrate a fast and powerful editor into your own programs. You get the source code, the manual and the know how.

Create your own word processor. We provide all the editing routines. You plug in the features you want. You could build a WordStar®-like editor with pull-down menus like Microsoft's® Word, and make it work as fast as WordPerfect™.

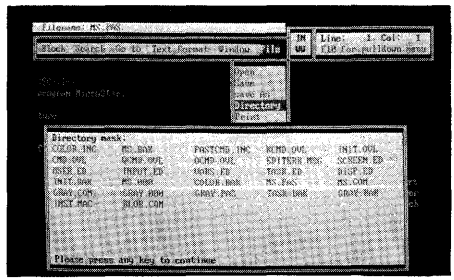
To demonstrate the tremendous power of Turbo Editor Toolbox, we give you the source code for two sample editors:

Simple Editor A complete editor ready to include in your programs. With windows, block commands, and memory-mapped screen routines.

MicroStar™ A full-blown text editor with a complete pull-down menu user interface, plus a lot more. Modify MicroStar's pull-down menu system and include it in your Turbo Pascal programs.

The Turbo Editor Toolbox gives you all the standard features you would expect to find in any word processor:

- Word wrap
- UNDO last change
- Auto indent
- Find and Find/Replace with options
- Set left and right margin
- Block mark, move and copy.
- Tab, insert and overstrike modes, centering, etc.



MicroStar's pull-down menus.

And Turbo Editor Toolbox has features that word processors selling for several hundred dollars can't begin to match. Just to name a few:

☑ ***RAM-based editor.*** You can edit very large files and yet editing is lightning fast.

☑ ***Memory-mapped screen routines.*** Instant paging, scrolling and text display.

☑ ***Keyboard installation.*** Change control keys from WordStar-like commands to any that you prefer.

☑ ***Multiple windows.*** See and edit up to eight documents—or up to eight parts of the same document—all at the same time.

☑ ***Multi-Tasking.*** Automatically save your text. Plug in a digital clock . . . an appointment alarm—see how it's done with MicroStar's "background" printing.

Best of all, **source code is included for everything in the Editor Toolbox.** Use any of the Turbo Editor Toolbox's features in your programs. And pay no royalties.

Minimum system configuration: The Turbo Editor Toolbox requires an IBM PC, XT, AT, 3270, PCjr or true compatible with a minimum 192K RAM, running PC-DOS (MS-DOS) 2.0 or greater. You must be using Turbo Pascal 3.0 for IBM and compatibles.

BORLAND INTERNATIONAL

Suggested Retail Price $69.95 (not copy-protected)

HOW TO BUY BORLAND SOFTWARE

NOT COPY-PROTECTED

BORLAND
INTERNATIONAL

For The Dealer Nearest You, Call (800) 556-2283

60 DAY MONEY-BACK GUARANTEE

To Order By Credit Card, Call (800) 255-8008

In California (800) 742-113

Notes:

Notes:

TURBO GAMEWORKS

Secrets And Strategies Of The Masters Are Revealed For The First Time

Explore the world of state-of-the-art computer games with Turbo GameWorks™. Using easy-to-understand examples, Turbo GameWorks teaches you techniques to quickly create your own computer games using Turbo Pascal®. Or, for instant excitement, play the three great computer games we've included on disk—compiled and ready-to-run.

TURBO CHESS

Test your chess-playing skills against your computer challenger. With Turbo GameWorks, you're on your way to becoming a master chess player. Explore the complete Turbo Pascal source code and discover the secrets of Turbo Chess.

"What impressed me the most was the fact that with this program you can become a computer chess analyst. You can add new variations to the program at any time and make the program play stronger and stronger chess. There's no limit to the fun and enjoyment of playing Turbo GameWorks' Chess, and most important of all, with this chess program there's no limit to how it can help you improve your game."

—George Koltanowski, Dean of American Chess, former President of the United Chess Federation and syndicated chess columnist.

TURBO BRIDGE

Now play the world's most popular card game—Bridge. Play one-on-one with your computer or against up to three other opponents. With Turbo Pascal source code, you can even program your own bidding or scoring conventions.

"There has never been a bridge program written which plays at the expert level, and the ambitious user will enjoy tackling that challenge, with the format already structured in the program. And for the inexperienced player, the bridge program provides an easy-to-follow format that allows the user to start right out playing. The user can "play bridge" against real competition without having to gather three other people."

—Kit Woolsey, writer and author of several articles and books and twice champion of the Blue Ribbon Pairs.

TURBO GO-MOKU

Prepare for battle when you challenge your computer to a game of Go-Moku—the exciting strategy game also know as "Pente"™. In this battle of wits, you and the computer take turns placing X's and O's on a grid of 19X19 squares until five pieces are lined up in a row. Vary the game if you like using the source code available on your disk.

Minimum system configuration: IBM PC, XT, AT, Portable, 3270, PCjr, and true compatibles with 192K system memory, running PC-DOS (MS-DOS) 2.0 or later. To edit and compile the Turbo Pascal source code, you must be using Turbo Pascal 3.0 for IBM PC and compatibles.

BORLAND INTERNATIONAL

4585 Scotts Valley Drive
Scotts Valley, CA 95066

ISBN 0-87524-146-8