TURBO PROLOG™ TOOLBOX

More than 80 tools and over 8,000 lines of source code—easily incorporated into programs!

It's the complete developer's toolbox and a major addition to Turbo Prolog

IBM® VERSION

PC, XT,® AT,® & True Compatibles

TURBO PROLOG TOOLBOX™

User's Guide and Reference Manual

BORLAND
INTERNATIONAL

First Edition
Printed in USA
9 8 7 6 5 4 3 2 1

Table of Contents

C H A P T E R

1

Introduction

Before the introduction of Turbo Prolog, Prolog was famous as the language for cost-effective software prototyping. With the implementation of Turbo Prolog, Prolog has become a practical tool that can be used for actual implementation work.

The Turbo Prolog Toolbox, a collection of software tools, increases the efficiency with which applications can be constructed. Many time-consuming (and therefore costly) operations—such as designing screen layouts—can be carried out quickly and easily using the predicates and programs defined within the Turbo Prolog Toolbox.

The Toolbox gives you over 80 tools (8,000 lines of source code) that you can incorporate into your programs. In addition, 40 sample programs demonstrate how to use the tools and can be adapted to your particular needs. These tools consist of built-in predicates, which specify relations among the defined objects.

Each major predicate is provided in a self-contained file that may be **included** in your programs. However, frequently used domains and predicate declarations have been incorporated into the files TDOMS.PRO and TPREDS.PRO, which should be **included** in all programs that use the tools described here. If your machine has limited memory and you need to reduce the amount of Turbo Prolog code, you can carefully edit the contents of TDOMS.PRO and TPREDS.PRO to delete the predicates you don't need. (It's a good idea to edit a *copy* of the files, so that you have the originals for later use.)

This manual assumes that you have a good working knowledge of Turbo Prolog. If you don't, you should first work through the tutorials in the *Turbo Prolog Owner's Handbook.*

What These Tools Can Do for You

The Toolbox is a distillation of the most useful tools for developing different types of software, including:

spreadsheets
database systems
file-handling systems
mailing-list systems
integrated systems
system utilities
communications packages
visual presentation systems
compilers
expert systems

Your Turbo Prolog Toolbox provides facilities for

- **User-interface design and construction** including a wide variety of menus (pull down, pop up, line, tree, and box), context-sensitive help, and predicates to include status lines in your programs.
- **Screen-layout design tools** that allow layout specifications to be constructed on the actual PC screen (and even on a virtual screen). Tool predicates can then make use of such specifications to generate printed reports.
- **Business graphics images** including predicates that create bar charts, pie charts, and graphs. An interface allows you to incorporate .PIC files into your Turbo Prolog programs.
- **Communication with remote devices** either through a serial link or a modem-to-modem connection. One of the sample programs using these predicates is a complete serial communications package; another allows file conversion on transmission to a printer.

- **Importing files from other systems** such as Reflex: The Database Manager, dBASE III, Lotus 1-2-3, and Symphony.
- **A parser generator** that automatically creates a Turbo Prolog source-code parser for a specified grammar. The parser can then be applied to construct your own user interfaces in Turbo Prolog.

How to Use This Book

Chapters 2 through 7 describe each tool in detail and list sample programs that show you how the tools can be used. While you'll want to read through all the chapters eventually, you can skip ahead to the chapters most relevant to your needs.

The final chapter, Chapter 8, is a reference guide. It's an alphabetical lookup of all of the tools, showing each tool's predicate, declaration, flow patterns, function, parameters, and any relevant information. It also contains a table that lists the Toolbox files and the files required by each file.

Following the chapters is an appendix, "Compiling a Project," that tells you how to load and run a project file. If you're unfamiliar with projects, you may want to read through this appendix early on and refer to it as needed.

You won't need to enter the code used to illustrate the concepts in each chapter; they're on your distribution disk under the file name given. They're also listed in the program examples that follow each explanation.

Who's Who?

In this manual, *user* always means the person who works with Turbo Prolog programs and *programmer* refers to the person who creates the Turbo Prolog code.

The Distribution Disks

The files containing the tools are on Distribution Disk 1. Disk 2 contains the example programs, which all start with an X.

The tool files on Disk 1 can be divided among four levels, each providing tools or declarations and clauses from which higher-level tools are constructed:

- **Level 1: Declaration files** These files at the lowest level contain declarations of domains used by the Toolbox predicates. For example, TDOMS.PRO is a declaration file.
- **Level 2: Low-level tool files** These contain several predicate definitions that are used by the higher level tool predicates. For example, TPREDS.PRO contains clauses for the predicates *repeat*, *max*, and *min*. GRAPHICS.OBJ contains external tool predicates, provided in .OBJ-code form because they have been implemented in C or assembler.
- **Level 3: Tool files** These files are function specific in that each contains either a major tool or a family of closely related tools. Thus, in order for an application to use two tools at this level (for instance, a status line and a menu), two tool files must be included in the application program (in this case, STATUS.PRO and MENU.PRO). The tool files include MENU.PRO, LONGMENU.PRO, BOXMENU.PRO, and SCRHND.PRO.
- **Level 4: Utility program files** These include a program for defining texts to be used when offering users context-sensitive help (HELPDEF.PRO), a program for defining screen layouts (SCRDEF.HLP), and a parser generator. (NOTE: The license agreement at the beginning of this manual specifies that these program files are for your personal use alone. You may not give away or sell them in any form.)

Tool Domains and Predicates

When using the tool predicates, it is assumed that the domains and predicates defined specifically for the purpose of constructing the tools are available in any containing program. These include the domains *stringlist*, *row*, *col*, *len*, *intergerlist*, and *key*, and many of the predicates in TPREDS.PRO. This availability is guaranteed if the appropriate files (TDOMS.PRO, TPREDS.PRO, and so on) are **included** in the containing program. Details of the predicates called by each tool predicate are easily obtained by listing the files that hold the tools.

If a tool file contains **database** declarations, these must be grouped with all the other **database** declarations in the containing application program.

Some tool predicates are implemented in C or assembler; they are provided in this Toolbox in the form of .OBJ files. For example, the tool predicate *loadpic,* which loads pictures into the Turbo Prolog system, is contained in the file PICTOOLS.OBJ. In order to use tools of this kind, you must create a Turbo Prolog project definition containing both PICTOOLS.OBJ and your program. If the project name is MYPROJ, then a corresponding module list, MYPROJ.PRJ, should be created containing the name(s) of the .OBJ file(s) used and the name(s) of your module(s). See Appendix A for more information on project files.

See the reference guide (Chapter 8) for a handy table listing all the Toolbox files and their contents.

Running the Sample Programs

All the sample programs are listed on the second distribution disk as Prolog source code. Their program names are prefixed with X. To execute most demos, load the program into the Prolog system and select the Run option from Turbo Prolog's main menu. Some of the demos (for instance, those concerning communications, graphics, and external data access tool predicates) need to be linked up to object modules. To execute these demos, select Compile Project from the Turbo Prolog main system menu and then give the name of the demo. (If you press Return at this point, a complete list of the demos implemented as projects is displayed on the screen.)

Installing the Toolbox on a Hard Disk

The demos are designed to be executed without modification. However, it is recommended that you set up a Toolbox directory. Here's one way to do it.

First, you should copy your Toolbox files into a subdirectory off of your Turbo Prolog directory. Let's say you named your Turbo Prolog directory PROLOG and will call the subdirectory TOOLBOX.

In Turbo Prolog, select the Setup option on the main menu. Then select Directories from the pull-down menu that appears. Set the directories as follows:

PRO directory	C:\TOOLBOX
OBJ directory	C:\TOOLBOX
EXE directory	C:\TOOLBOX
TURBO directory	C:\PROLOG

That's it—you're set to begin programming with the Toolbox.

Installing the Toolbox on Floppy Disks

If you have a computer with two floppy drives, you first need to set up a Turbo Prolog work disk (disk 1) using a blank formatted disk. Insert your Turbo Prolog disk into Drive A and disk 1 into Drive B. Copy the following files onto disk 1:

From the Turbo Prolog Program Disk:
PROLOG.OVL
PROLOG.HLP
PROLOG.ERR
PROLOG.SYS
Do not copy PROLOG.EXE.

From the Turbo Prolog Library & Sample Programs Disk:
PROLOG.LIB
INIT.OBJ

From your DOS Disk:
COMMAND.COM

Remove disk 1 from Drive B.

Next, insert a second blank formatted disk (a development work disk, disk 2) into Drive B and copy these Toolbox files onto it:

TDOMS.PRO
TPREDS.PRO

(You'll need to do this for each development work disk you set up.) Then copy the tools and demo programs required for the program or chapter you are working on. For example, to run the status line example, you should copy STATUS.PRO and XSTATUS.PRO onto disk 2. Remember, all files are listed at the beginning of the Reference Guide.

Now, load Turbo Prolog (the disk containing PROLOG.EXE and PROLOG.OVL) from Drive A. Remove your Turbo Prolog disk and insert disk 1 into Drive A.

Select Setup from the main menu and Directories from the pull-down menu. Set up the following directories:

PRO directory:	B:\
OBJ directory:	B:\
EXE directory:	B:\
TURBO directory:	A:\

You are now ready to use the Turbo Prolog Toolbox.

A Helpful Hint

Start with the simplest example in a chapter and add files to your work disk as needed. When you've filled your development work disk, you can select unnecessary files and delete them. That way, you can continue using the same work disk.

As you progress, you may find it convenient to set up different development work disks according to your areas of interest.

Reserved Windows

The Turbo Prolog tools use windows 80 to 85 inclusive. These windows can not be used by any program that uses the tool predicates.

P A R T

1

User's Guide

C H A P T E R

2

User Interface Tools

This chapter introduces the various tools that help you develop customized user interfaces. It includes tools for status lines, different types of menus, line inputs, context-sensitive Help, window resizing, and some BIOS (Basic Input/Output System) calls. Sample programs follow each tool description, to suggest how you can implement the tool in your program.

Status Lines

STATUS.PRO contains a set of tools that uses window number 83 (see *makewindow*) to display a status line at the bottom of the screen, rather like the Turbo Prolog editor status line, which reminds you of the action of each function key. A status line is initially displayed via the *makestatus* tool predicate, which takes the form

```
makestatus(ATTR,STRING)
```

ATTR is the window color attributes; STRING is the text status window. The call

```
makestatus(112,"d=date t=time m=more l=less ")
```

reminds users that you press *d* for date, and so on.

If a containing program destroys the status line, such as by drawing a window that overlaps it, the status line can be refreshed with the tool predicate *refreshstatus*. The tool predicate *removestatus* deletes the status window. Another tool predicate, *changestatus,* takes the string and is used with the call *changestatus(newstatus).*

If a call to *makestatus* is followed by calls that can fail, *tempstatus* should be used instead. Then, in the event of backtracking to the *tempstatus* call, the status line is removed.

```
tempstatus (ATTR,STRING)
```

A Sample Program Using a Status Line

The following program, XSTATUS.PRO, demonstrates the status-line family of tool predicates. Remember, you can refer to the table in the reference guide to quickly see what files XSTATUS.PRO requires.

XSTATUS.PRO

```
include "tdoms.pro"
include "tpreds.pro"
include "status.pro"

goal
   makewindow(2,7,0,"",0,0,25,80),
   makewindow(1,7,7,"",0,0,24,80),
   cursor(20,0),
   makestatus(7,"Press any key  (first status line)"),
   write("A status line can be created easily with the makestatus
         tool predicate."),
   nl,nl,
   readkey(_),
   write("*******************************************************\n"),
   write("\nThe contents of the status line can later be changed by the"),
   write("\nchangestatus command"),
   write("\n\nPress the space bar to see the effect of \n"),
   write("\tchangestatus(\"A new status line: please press the spacebar\")\n\n"),
   readkey(_),
   changestatus("A new status line: please press the spacebar"),
   readkey(_),
   changestatus("A new status line"),
   write("*******************************************************\n"),
   write("\nA status line can be removed by the removestatus predicate."),
   write("\n\nPress the spacebar to see the effect of removestatus. When"),
   write("\nyou are ready to continue, press the spacebar once again\n\n"),
   readkey(_),
removestatus,
readkey(_),
write("*************************************************\n"),
write("\nAnd they can overlap. Press the space bar again"),nl,nl,
readkey(_),
write("We will create two status lines, one on top of another"),nl,
write("using:\n\n\t\tmakestatus(7,\"First of two overlapping status
      lines\"),"),nl,
write("\t\tmakestatus(135,\"Second of two overlapping status
      lines\"),"),nl,nl,nl,
makestatus(7,"First of two overlapping status lines"),
makestatus(135,"Second of two overlapping status lines"),
write("Below is the second status line. Press the space bar to see
      the first"),
write("\nand then press the space bar once again to remove this also.\n"),
readkey(_),
removestatus,
readkey(_),
removestatus,
write("\n*************************************************\n"),
write("\nIf you want the statusline to disappear automatically"),nl,
write("during fail, then you should use tempstatus instead of
      makestatus"),nl,
readkey(_),
tempstatus(112," This line is created by tempstatus"),
write("\n*************************************************\n"),
write("\nSometimes the statusline can disappear because another window"),
write("\nis placed on the top of it."),
write("\n\nPress any key and the statusline will disappear"),
write("\nbecause of: \"shiftwindow(2),shiftwindow(1)\"\n"),
readkey(_),
shiftwindow(2),shiftwindow(1),
write("\n\nPress any key to see the effect of refreshstatus\n"),
readkey(_),
refreshstatus,
write("\n*************************************************\n"),
write("\n\nNow the program will fail. Notice that tempstatus"),
write("\nremoves the status line."),
readkey(_),
fail.
```

Some Basic Menus

This section describes three basic menu tools—*menu, menu_leave,* and *menu_mult*—which are in the file MENU.PRO.

The *menu* Tool Predicate

The *menu* predicate implements a menu in which the arrow keys can be used to indicate the various options and the F10 or Return key to select an option. It takes the form

```
menu(PosRow,PosCol,Wattr,Fattr,ItemList,Title,InitItem,ChoiceCode)
```

and has eight parameters:

- *PosRow* and *PosCol*, respectively, indicate the row and column positions on the text screen of the upper left-hand corner of the menu window.
- *Wattr* and *Fattr* specify the attribute values for the inside of the window and its frame; if the frame attribute is zero, the window will not have a frame.
- *Itemlist* is a list of strings, one string for each menu item, with a description of that item.
- *Title* is a string heading for the menu, which is included in the menu window's frame when it is drawn.
- *InitItem* specifies the number of the menu item on which the selection bar is drawn when the menu is first displayed.
- *ChoiceCode,* the integer parameter, is bound to a code number indicating the user's actual menu selection:

0	Esc was pressed and no selection made.
1	The first item on the menu was selected.
2	The second item on the menu was selected.
3	The third item on the menu was selected, and so on.

Thus, the call

```
menu(5,10,7,7,[Basic,Pascal,Lisp,Prolog,Modula2],
     "Which is your favorite language?",4,Langno)
```

creates a window positioned at row 5, column 10 of the text screen, as shown in Figure 2.1. It contains white text on a black background with a white frame.

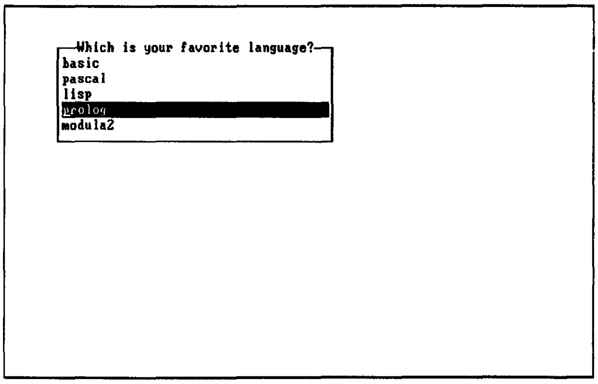


Figure 2.1: A Call to the *menu* Tool Predicate

The word `prolog` is highlighted when the menu is first displayed. If the user presses the arrow keys to highlight (`modula2`) and then presses Return, *menu* succeeds with *Langno* instantiated to 5.

Using *menu* in a Sample Program

The following sample program uses *menu* to construct a menu-based career advisory system that suggests the best career for its user. In a fairly simple-minded way, it works out which of three careers best suits the user, given an analysis of the user's personality. The program is in the file XCAREER.PRO, whose required files are listed in the files table at the beginning of the reference guide.

XCAREER.PRO

```
include "tdoms.pro"
include "tpreds.pro"
include "menu.pro"

domains
   numberlist = integer*

predicates
   ask(numberlist,integer)
   career(integer,string)
   q(integer,string,stringlist)

goal
   makewindow(1,7,0,"",0,0,25,80),
   ask([1,2,3,4],Total),career(Total,Job),
   makewindow(1,7,7,"Career",20,10,5,50),
   nl,write("You should be in",Job),readchar(_).

clauses
   q(4,"If you were at a party and saw someone standing alone, would you:",
         ["organize lots of people to go over and talk to that person",
          "go over and talk to him or her yourself",
          "just smile at him or her sympathetically",
          "take no notice"]).

   q(3,"Which of the following activities do you most enjoy?",
         [parties,"going out for a meal",sports,cinema,"flower arranging",
          "reading a book"]).

   q(2,"Which one of the following best describes you?",
         [obstinate,determined,ambitious,considerate,thoughtful,
          "couldn't care what turns up"]).

   q(1,"Which one of the following best describes you?",
         [wild,"very extroverted",extroverted,happy,"quiet and refined",
          quiet,"very introverted","extremely shy"]).

   ask([],0).
   ask([X | Y],S) :-
         ask(Y,T),q(X,Title,Choicelist),
         menu(10,7,7,7,Choicelist,Title,1,ChoiceNo),
         S = T+ChoiceNo.

   career(Total,advertising) :- Total<8.
   career(Total,computing) :- Total>7,Total<20.
   career(_,archaeology).
```

The Tool Predicate *menu_leave*

This tool is also found in the file MENU.PRO and implements the predicate *menu_leave,* which creates a pop-up menu as in *menu.* However, on return from *menu_leave,* the window is *not* removed from the screen. *menu_leave* takes the same eight parameters as *menu.*

Using *menu_leave* in a Sample Program

If the clauses defining the *ask* predicate of the previous sample program are replaced with a version in which *menu_leave* is used instead of *menu,* successive windows remain on the screen and overlap. (This amended program is in XLCAREER.PRO.)

XLCAREER.PRO

```
ask([],0).
ask([X | Y],S) :- ask(Y,T),q(X,Title,Choicelist),
      R = 4*(4-X), C = 7*(4-X), FATTR = 7+16*X
      menu_leave(R,C,7,FATTR,Choicelist,Title,1,ChoiceNo),
      S = T+ChoiceNo.
```

The Tool Predicate *menu_mult*

The clauses that define *menu_mult* are in the file MENU.PRO. *menu_mult* implements a pop-up menu similar to the *menu* predicate but returns a *list* of selections made from the menu rather than just a single selection. This list consists of numeric codes for the selections made (as with *menu* and *menu_leave*). (If Esc is pressed during menu selection, however, an empty list is returned.) Use the arrow keys to highlight a choice from the menu and press Return to select it. Press F10 to indicate that all the desired selections have been made.

The parameters for *menu_mult* are the same as *menu,* except that the choice parameter is an INTEGERLIST rather than an INTEGER, and its definition takes the form

InitItem is an intgrlist rather than an integer

```
menu_mult(ROW,COL,ATTR,ATTR,STRINGLIST,STRING,INTEGERLIST,INTEGERLIST)
```

and a call of the form

```
menu_mult(Row,Col,Wattr,Fattr,List,Header,StartList,ChoiceList)
```

places a menu on the screen, with these parameters:

- *Row* and *Col* indicate the row and column positions on the text screen of the top left-hand corner of the menu window.
- *Wattr* and *Fattr* specify the attribute values for the window forming the menu and the menu frame.
- *List* names the items on the menu.
- *Header* is the string holding the menu title.
- *StartList* specifies the selection to be highlighted when the menu is first displayed.
- *ChoiceList* returns the selections made from the menu.

Using *menu_mult* in a Sample Program

The following sample program (in the file XIQ.PRO) uses *menu_mult* to create a simple multiple-choice brain-teaser.

XIQ.PRO

```
include "tdoms.pro"
include "tpreds.pro"
include "menu.pro"

domains
   numberlist = integer*

predicates
   solution(integer,integerlist)
   ask(numberlist,integer)
   append(integerlist,integerlist,integerlist)
   member(integer,integerlist)
   q(integer,string,stringlist)
   same(integerlist,integerlist)
   allelts(integerlist,integerlist)
   message(integer)
   checkanswer(integer,integerlist,integerlist,integer)

goal
   makewindow(1,7,7,"Professor Blanketbrain's Intelligence Test",0,0,25,80),
   write("\n\n\tMake any number of selections using the RETURN key."),
   write("\n\tWhen all selections have been made, press F10."),
    nl,nl,ask([4,3,2,1],Totalok),
    nl,nl,write("     - The puzzle is over. "),nl,nl,
    message(Totalok),nl,nl,
    write("    Please press the space bar."),
    readchar(_).
```

clauses

```
    q(1,"Question 1: Which animal(s) are the odd ones out?",
          [dog,cat,duck,rabbit,whale,swan]).

    q(2,"Question 2: Which number(s) are the odd ones out?",
          ["2","3","4","5","6","7" ]).

    q(3,"Question 3: Which colors are the odd ones out?",
          [red,orange,yellow,pink,green,brown,black]).

    q(4,"Question 4: Which name(s) are the odd ones out?",
          [kim,tom,george,alison,mary,martha]).

    solution(1,[3,6]).
    solution(2,[3,5]).
    solution(3,[4,6,7]).
    solution(4,[1,2]).

    message(4) :-
          write("    You got all 4 correct."),nl,
          write("    Professor Blanketbrain thinks you are a genius").

    message(0) :-
          write("    You got none of them correct."),nl,
          write("    Professor Blanketbrain thinks you can improve your knowledge").

    message(Totalok) :-
          write("    You got ",Totalok," correct. "),nl,
          write("    Not bad but not good either.").

    ask([],0).
    ask([X | Y],Newscore) :-
          ask(Y,Oldscore),q(X,Title,Choicelist),
          menu_mult(10,10,7,7,Choicelist,Title,[],Oddlist),
          solution(X,L),checkanswer(X,L,Oddlist,Score),
          Newscore = Oldscore+Score.

    checkanswer(X,L,Oddlist,1) :-
          same(L,Oddlist),
          write("    Your answer to question  ",X is),
          write(" correct. Please press the space bar."),
          nl,readchar(_).
    checkanswer(X,_,_,0) :-
          write("    Sorry, but your answer to question ",X is),
          write(" wrong  - Please press the space bar."),
          nl,readchar(_).

    same(A,B) :- allelts(A,B),allelts(B,A).

    allelts([],_).
    allelts([X | Y],B) :- member(X,B),allelts(Y,B).
```

```
member(X,[X|_]).
member(X,[_|Y]) :- member(X,Y).

append([],X,X).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

Using *menu*, *menu_leave*, and *menu_mult* in a Program

The program XMENU.PRO demonstrates the use of menus in a program. It asks the user to choose one or more numbers from each selection offered and displays the choices made. Menus of various designs are produced—with and without frames, with different highlights, and so on. In the program, *CR* stands for Carriage Return (the Return key).

XMENU.PRO

```
include "tdoms.pro"
include "tpreds.pro"
include "status.pro"
include "menu.pro"

predicates
   test

clauses
   test :-
         changestatus("Normal menu    CR:Select  ESC:Abort"),
         menu(10,10,7,7,[first,second,third,fourth],"Normal menu",3,CHOICE),
         write("You chose: ",CHOICE),nl,fail.

   test :-
         changestatus("Normal menu with specified initial selection
                      CR:Select
                      ESC:Abort"),
         menu(10,10,7,7,[first,second,third,fourth],"Normal menu",3,CHOICE),
         write("You chose: ",CHOICE),nl,fail.

   test :-
         changestatus("Use of menu without a frame    CR:Select   ESC:Abort"),
         menu(10,10,112,0,[first,second,third,fourth],"No frame",2,CHOICE),
         write("You chose: ",CHOICE),nl,fail.

   test :-
         changestatus("Use of menu_leave    CR:Select   ESC:Abort"),
         menu_leave(5,0,7,7,[first,second,third,fourth],first,1,CH1),
         menu_leave(10,20,7,7,[first,second,third,fourth],second,2,CH2),
         menu_leave(15,40,7,7,[first,second,third,fourth],third,3,CH3),
        removewindow,removewindow,removewindow,
        write("You chose: ",CH1,", ",CH2,", ",CH3),nl,fail.

   test :-
        changestatus("Use of menu_mult "(F10:End CR:Select or delete
                     ESC:Abort"),
        menu_mult(10,10,7,66,[first,second,third,fourth,fifth,"6","7"],
                 "test",[3,6]CHOICEL),
        write("You chose: ",CHOICEL),nl,fail.

goal
   makewindow(1,23,0,"test",0,0,24,80),
   makestatus(112,"Choose a number"),
   test,
   changestatus("End of demo"),nl.
```

longmenu, longmenu_leave, and *longmenu_mult*

The tools just discussed, *menu, menu_leave,* and *menu_mult,* are limited to a maximum of 23 items on any menu—that's 25 lines on the screen, less 2 for the border drawn around the menu. The predicates *longmenu, longmenu_leave,* and *longmenu_mult* in the file LONGMENU.PRO, on the other hand, allow arbitrarily long lists of menu items. They behave like their more restrictive counterparts except that each has an extra parameter after the ROW and COL parameters. This parameter determines the number of menu items to be displayed on the screen at any one time; the remaining choices are displayed by using the cursor keys, PgUp, and PgDn. For example, the parameters of *longmenu* are given by

```
   longmenu(Row,Col,Maxrows,Wattr,Fattr,Stringlist,Header,StartChoice,Selection)
```

Using the *longmenu* Tool Predicates in a Program

The file XLONGMNU.PRO contains the following program, which requests its user to make one or more selections from a list of 50 alternatives (using the up and down arrow keys to view them), and then displays the choice or choices made. The user is also informed as to whether *longmenu, longmenu_leave,* or *longmenu_mult* was applied to construct the part of the program that he or she has just used.

XLONGMNU.PRO

```
include "tdoms.pro"
include "tpreds.pro"
include "longmenu.pro"
include "status.pro"

predicates
   test
   for(INTEGER,INTEGER,INTEGER)
   str(STRING)

clauses
   for(X,X,_).
   for(I,A,B) :- B>A,A1 = A+1,for(I,A1,B).

   str(first).
   str(second).
   str(third).
   str(S) :- for(I,4,50),str_int(S,I).

   test :-
         changestatus("Longmenu is used to select from more than
                      23 alternatives    CR:Select    ESC:Abort"),
         findall(X,str(X),L),
         longmenu(10,10,5,7,7,L,"longmenu",0,CHOICE),
         write("Last time you chose: ",CHOICE),nl,fail.

   test :-
         changestatus("longmenu with pre-selected starting point for the cursor
                      CR:Select    ESC:Abort"),
         findall(X,str(X),L),
         longmenu(10,10,5,7,7,L,"longmenu",3,CHOICE),
         write("Last time you chose: ",CHOICE),nl,fail.

   test :-
         changestatus("Using longmenu to create an unframed menu
                      CR:Select    ESC:Abort"),
         findall(X,str(X),L),
         longmenu(10,10,5,112,0,L,"This message will not be visible",2,CHOICE),
         write("Last time you chose: ",CHOICE),nl,fail.

   test :-
         changestatus("Longmenu_leave: Make one selection from each menu
                      displayed   CR:Select    ESC:Abort"),
         findall(X,str(X),L),
         longmenu_leave(5,0,5,7,7,L,first,1,CH1),
         longmenu_leave(10,20,5,7,7,L,second,2,CH2),
         longmenu_leave(15,40,5,7,7,L,third,3,CH3),
         removewindow,removewindow,removewindow,
         write("Last time you chose: ",CH1,", ",CH2,", ",CH3),nl,fail.

   test :-
         changestatus("Use longmenu_mult to allow any number of options
                      F10:End   CR:(Un)select    ESC:Abort"),
        findall(X,str(X),L),
        longmenu_mult(10,10,5,7,7,L,"test",[2],CHOICEL),
        write("Last time you chose: ",CHOICEL),nl,fail.

goal
   makewindow(1,7,0,"test",0,0,24,80),
   write("Use cursor keys to inspect the rest of the menu items\n\n"),
   makestatus(112,"Choose a number"),
   test,
   write("End of demo").
```

Box Menus

With a box menu, users can select items from more than one column of choices. The Turbo Prolog system, for example, displays a box menu containing file names when the empty string is given as a response to the *Load* option of *Files*. The ***boxmenu*** tool predicate is contained in the file BOXMENU.PRO and takes the form

```
   boxmenu(Row,Col,NoOfRows,NoOfCols,Wattr,Fattr,Stringlist,Header,
          StChoice,Choice)
```

where

- *Row* and *Col* determine the position of the window.
- *NoOfRows* determines the number of rows in the window.
- *NoOfCols* determines the number of columns in the window.
- *Wattr* and *Fattr* determine respectively the attributes for the window itself and for its frame—if *Fattr* is zero, there is no frame on the window.
- *Stringlist* contains the list of possible selections.
- *Header* is the text in the top of the window.
- *StChoice* determines where the bar should be placed when the menu is first displayed.
- *Choice* is bound to the selection made.

For example,

```
                40,7,
   boxmenu(5,5,7,40,7,5,[a,b,c,d,e,f,g,h,i,j,k,l,m,n],"letters",3,CHOICE)
```

displays a menu at row 5, column 5 of the screen, which occupies 7 rows and 40 columns in total—including the frame, since *Fattr* is non-zero. The items on the menu are the letters *a,b,c,...,n,* and the menu is headed with the string *letters*. When the box menu is first displayed, the third menu entry is highlighted, and the letter selected is returned in the variable *Choice*.

BOXMENU.PRO also contains its own corresponding versions of *menu_leave* and *menu_mult*. *boxmenu_leave* and *boxmenu_mult* are related to *menu_leave* and *menu_mult* (respectively) in the same way that *boxmenu* and *menu* are related. *boxmenu_leave* takes the same parameters as *boxmenu*, whereas *boxmenu_mult* takes the form

```
boxmenu_mult(Row,Col,NoOfRows,NoOfCols,Wattr,Fattr,Stringlist,Header,
        StChoiceList,ChoiceList)
```

The only difference is that the parameter in which the selections made are returned is now a list of INTEGERs, and the starting items are highlighted according to the integerlist St List Choice,

Using *boxmenu* and *boxmenu_mult* in a Program

Once again, the sample program (in XBOXMENU.PRO) involves selecting numbers in different ways. First, one selection each must be made from three box menus, and the choices are then displayed. Next, as a demonstration of *boxmenu_mult*, several choices can be made from one menu.

XBOXMENU.PRO

```
include "tdoms.pro"
include "tpreds.pro"
include "boxmenu.pro"
include "status.pro"

predicates
   test
   for(INTEGER,INTEGER,INTEGER)
   str(STRING)

clauses
   for(X,X,_).
   for(I,A,B) :- B>A,A1 = A+1,for(I,A1,B).

   str(one).
   str(two).
   str(three).
   str(S) :- for(I,4,250),str_int(S,I).
   test :-
        findall(X,str(X),L),
        boxmenu(0,0,18,78,7,7,L,"This is a demonstration of boxmenu",
              0,CHOICE),
        write("Last time you chose the number:",CHOICE),nl,fail.

   test :-
        changestatus("Now choose three numbers, one from each menu"),
        findall(X,str(X),L),
        boxmenu_leave(0,0,10,50,7,7,L,
              "This is a demonstration of boxmenu_leave",0,CH1),
        boxmenu_leave(4,20,10,50,7,7,L,
              "This is a demonstration of boxmenu_leave",0,CH2),
        boxmenu_leave(8,40,10,50,7,7,L,
              "This is a demonstration of boxmenu_leave",0,CH3),
        removewindow,removewindow,removewindow,
        write("Last time you chose the numbers: ",CH1,", ",CH2,", ",CH3),
        nl,fail.

   test :-
        changestatus("Now choose several numbers
              F10:End   CR:Select or remove"),
        findall(X,str(X),L),
        boxmenu_mult(0,0,18,78,7,7,L,
              "This is a demonstration of boxmenu_mult",
              [5,10,20],CHOICEL),
        write("Last time you chose the numbers:",CHOICEL),nl,fail.

goal
   makewindow(1,7,0,"",0,0,25,80),
   makewindow(1,7,0,"test",20,0,4,80),
   makestatus(112,"Choose a number"),
   test,
   write("End of demonstration"),nl,readkey(_).
```

A Line Menu

The tool predicate *linemenu* implements a menu on a single line and is contained in the file LINEMENU.PRO. It takes the form

```
linemenu(Row,Attr,Attr,Stringlist,Integer)
```

which has five parameters:

- The *Row* on which the line menu is to appear.
- The two attributes of the window containing the menu and its frame—both of internal type *Attr*.
- A *Stringlist* of the items on the menu.

- A variable of *Integer* type that will become instantiated to an integer code for choices made in the usual way.

Thus,

```
linemenu(0,7,0,[run,compile,edit,options,files,
        setup,quit],CHOICE)
```

produces a line menu rather like the top line of Turbo Prolog's pull-down menu system.

Using *linemenu* in a Program

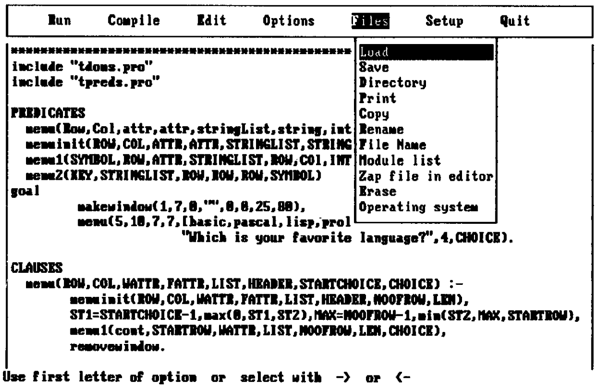
This program, in the file XLINEMNU.PRO, asks the user to select a programming language, first from a *framed* line menu and then from an *unframed* line menu. It then displays the choice made in each case.

XLINEMNU.PRO

```
include "tdoms.pro"
include "tpreds.pro"
include "linemenu.pro"
include "status.pro"

goal
   makewindow(1,7,0,"test",0,0,24,80),
   cursor(5,0),
   makestatus(112,"Choose a programming language "),
   linemenu(0,7,7,["Basic","Fortran","Pascal","Apl","Lisp","Prolog"],CH1),
   write("Last time you chose item number ",CH1),nl,
   changestatus("Now choose from an unframed menu"),
   linemenu(0,7,0,["Basic","Fortran","Pascal","Apl","Lisp","Prolog"],CHOICE),
   changestatus(""),
   write("This time you chose item number ",CHOICE),nl,
   write("End of demo"),nl,readkey(_).
```

Pull-down Menus

PULLDOWN.PRO contains pull-down menus that allow menu items to be grouped into related families, as in the Turbo Prolog user interface. Once you move the cursor to an item in the main menu line and then press Return, another menu appears, pulled down vertically below the horizontal menu item (see Figure 2.2). This vertical menu contains items closely related to the horizontal heading. Of course, some horizontal menu items have no corresponding vertical menu (such as Run, Compile, and Edit in Turbo Prolog's main system menu).



```
*********************************************
include "tdoms.pro"
include "tpreds.pro"

PREDICATES
  menu(Row,Col,attr,attr,stringList,string,int
  menuinit(ROW,COL,ATTR,ATTR,STRINGLIST,STRING
  menu1(SYMBOL,ROW,ATTR,STRINGLIST,ROW,COl,INT
  menu2(KEY,STRINGLIST,ROW,ROW,ROW,SYMBOL)
goal
        makewindow(1,7,0,"",0,0,25,80),
        menu(5,10,7,7,[basic,pascal,lisp,prol
                        "Which is your favorite language?",4,CHOICE).

CLAUSES
  menu(ROW,COL,WATTR,FATTR,LIST,HEADER,STARTCHOICE,CHOICE) :-
       menuinit(ROW,COL,WATTR,FATTR,LIST,HEADER,NOOFROW,LEN),
       ST1=STARTCHOICE-1,max(0,ST1,ST2),MAX=NOOFROW-1,min(ST2,MAX,STARTROW),
       menu1(cont,STARTROW,WATTR,LIST,NOOFROW,LEN,CHOICE),
       removewindow.

Use first letter of option  or  select with  ->  or  <-
```

Figure 2.2: A Sample Pull-down Menu

The *pulldown* tool predicate in PULLDOWN.PRO implements a pull-down menu of this type. The pull-down window is declared as follows:

```
pulldown(Attr,Menulist,Choice,SubChoice)
```

Its parameters are given by

```
pulldown(Attr,Menulist,Choice,SubChoice)
```

where

- *Attribute* is the color attribute to be used in all the windows (and their frames) that are part of the pull-down menu system.
- *Menulist* is a list constructed using the *curtain* functor (as described in the following paragraphs), which contains the text for each of the menus that can be pulled down.
- On return from *pulldown, Choice* is bound to the code number for the horizontal menu item on which the cursor was resting when a selection was made.

- On return from *pulldown, SubChoice* is bound to the code number for the actual menu selection, made by pressing Return.

(If there is no vertical menu corresponding to a horizontal menu item and that horizontal menu is selected by pressing Return, *SubChoice* is bound to zero and *Choice* is bound to the code number for the horizontal menu item.)

The following parameters create a main pull-down menu similar to Turbo Prolog's main menu:

```
pulldown (7,
         [curtain(5,"Run",[]),
          curtain(13,"Compile",[]),
          curtain(25,"Edit",[]),
          curtain(35,"Option",["Memory","Obj File","Exe File"]),
          curtain(47,"File",["Load","Save","Directory","Print"]),
          curtain(56,"Setup",["Color","Window size","DIR"]),
          curtain(67,"Quit",[])],
         Choice,Sub_Choice),
```

In a pull-down menu system built using *pulldown,* the action taken when a given menu item has been selected is determined by the *pdwaction* predicate. This allows the program to perform commands with the pull-down window remaining on the screen. If *pdwaction* fails, *Choice* and *SubChoice* are bound to the relevant values; the entire menu system is removed from the screen and *pulldown* succeeds. If, on the other hand, *pdwaction* succeeds, then *pulldown* continues to loop and the last selected menu item remains highlighted.

To use *pulldown,* you must include the database predicate *pdwstate* in your program's **database** declarations as follows:

```
database
   pdwstate(ROW,COL,SYMBOL,ROW,COL)
   ......
```

Menulist is specified by giving a list of values for the *curtain* functor. It takes three parameters, as in

```
   curtain(COL,STRING,STRINGLIST)
```

The first parameter specifies which column the horizontal menu item should begin on, the second gives the name of that menu item, and the third is the list of items that are to appear in the corresponding vertical menu. Thus,

```
   curtain(4,"Animals",["Dog","Cat","Bullfinch"])
```

specifies part of a pull-down menu system in which, if the *Animals* heading is selected, a submenu appears containing *Dog, Cat,* and *Bullfinch.*

A simple model showing part of Turbo Prolog's user interface can be constructed using *pulldown* with the following MENULIST:

```
[curtain(7,"Run",[]),
 curtain(14,"Compile",[]),
 curtain(26,"Edit",[]),
 curtain(35,"Options",["Memory","Obj File",Exe File"]),
 curtain(47,"Files",["Load","Save","Directory","Print"]),
]
```

When the user makes a choice, any actions to be carried out must be specified by a clause with the *pdwaction* predicate. This clause takes just two integer parameters: The first gives the code of the horizontal (or main) menu item, and the second specifies the corresponding vertical menu item selected (which is zero if no vertical menu exists). Thus, if *pulldown* were used with the above MENULIST and the Save option of the Files menu were selected, then the corresponding action would be defined by completing the right-hand side of the clause:

```
pdwaction(5,2) :- ............
```

Besides pressing Return, the user can also select a given item from a pull-down menu by pressing the highlighted letter of that item. *pulldown* highlights the first capital letter of each menu item or, if no letters are capitalized, the first lowercase letter.

Using *pulldown* in a Program

The following sample program from XPULLDW.PRO displays simple messages on the screen. If horizontal menu item 5 (*Quit*) is selected, however, the action is determined by

```
pdwaction(5,0) :- msg(15,20,"Please press the space bar"),exit.
```

which displays the message *Please press the space bar* at row 15, column 20. The standard predicate *exit* then returns control to the Turbo Prolog system.

After running this program as supplied on the Toolbox disk, you may find it revealing to replace the above *pdwaction* clause with the following:

```
pdwaction(5,0) :- fail.
```

If the program is then re-run and the *Quit* menu option selected, you'll see the action of *pulldown* when *pdwstate* fails.

XPULLDW.PRO

```
include "tdoms.pro"

database
   pdwstate(ROW,COL,SYMBOL,ROW,COL)

include "tpreds.pro"
include "status.pro"
include "pulldown.pro"

predicates
   msg(ROW,COL,STRING)

clauses

/* Pulldown tries first to call pdwaction with the windows open. */
/* If no pdwaction is found CHOICE and SUBCHOICE are returned and */
/* the windows are removed. */

   pdwaction(1,1) :- msg(3,10,"Input 1 selected").
   pdwaction(1,2) :- msg(4,10,"Input 2 selected").
   pdwaction(1,3) :- msg(5,10,"Input 3 selected").
   pdwaction(2,0) :- msg(5,25,"List all").
   pdwaction(3,1) :- msg(3,42,"Load something").
   pdwaction(3,2) :- msg(4,42,"Save something").
   pdwaction(3,3) :- msg(5,42,"Delete some garbage").
   pdwaction(3,4) :- msg(6,42,"Change directory").
   pdwaction(4,1) :- msg(3,45,"Actions with directories").
   pdwaction(4,2) :- msg(4,45,"Change colors").
   pdwaction(5,0) :- msg(15,10,"Please press the space bar."),exit.

   msg(R,C,S) :-
         makewindow(1,7,7,"",R,C,5,30),
         window_str(S),
         readkey(_),
         removewindow.

goal

/*
          1         2         3         4         5         6         7
01234567890123456789012345678901234567890123456789012345678901234567890123456789
     INPUT          LIST           FILES         SETUP          QUIT
*/
   makewindow(1,7,0,"",0,0,25,80),
   pulldown(7,
            [curtain(5,"Input",["First","Second","Third"]),
             curtain(20,"List",[]),
             curtain(36,"Files",["Load","Save","Delete","directory"]),
             curtain(50,"Setup",["Directories","Colors"]),
        curtain(65,"Quit" ,[])
      ],CH,SUBCH ),
    write("\n    CH = ",CH),
    write("\n SUBCH = ",SUBCH),nl.
```

Tree Menu Input

The file TREE.PRO includes tools to build tree menus. A tree menu may come in handy when your underlying data structure is a tree—for example, in designing an expert-system shell or even a customized expert system. Expert systems are merely databases filled with detailed facts on a specific subject, which are then manipulated by an inference engine. The *treemenu* tool predicate takes three arguments:

```
treemenu(SYMBOL,TREE,SELECTOR)
```

A SYMBOL can be *up, down, left,* or *right* to indicate which way the tree is to be drawn when it is displayed. More details follow on the TREE definition. The SELECTOR parameter returns a code for the menu item chosen.

In each case, the code to be returned when a certain tree item is selected is included in the definition of that item via the *tree* functor. This functor takes three arguments: The first is a STRING giving the text for that menu item; the second is the code to be returned if that menu item is selected; and the third is a TREELIST specifying the rest of the tree. Thus, a TREE domain is defined by

```
SELECTOR = INTEGER
TREE = tree(STRING,SELECTOR,TREELIST)
TREELIST = TREE*
```

The SELECTOR can actually be of any type you prefer.

You select from a menu with the arrow keys and the Return or F10 key as usual.

When a selection has been made, the predicate *treeaction* is called with the SELECTOR as the parameter. This makes it possible to perform an action with the tree remaining on the screen. (See also *pdwaction* for the pull-down menu.)

Using *treemenu* in a Program

The following sample program (from XTREE.PRO) illustrates how *treemenu* is used.

XTREE.PRO

```
code=3000

include "tdoms.pro"

domains
   SELECTOR = INTEGER
   TREE = tree(STRING,SELECTOR,TREELIST)
   TREELIST = TREE*

database
   treewindow(ROW,COL,~~ROW,COL,ROW,COL~~)
   treechoice(SELECTOR)
```

To use the tree window tool, you must include the data base declarations

```
include "tpreds.pro"
include "status.pro"
include "menu.pro"
include "tree.pro"

clauses
   treeaction(_) :-
         cursor(R,C),
         menu(R,C,7,7,[continue,select],test,0,CH),
         CH = 2.

predicates
   test
   show(SYMBOL)

clauses
   show(DIRECTION) :-
         treemenu(DIRECTION,tree("start",1,
                  [tree("start_problem",2,
                        [tree("gasoline_help",3,[]),
                         tree("electrical_system",4,
                               [tree("battery_flat",5,
                                     [tree("another",6,[]),
                                      tree("recharge",7,[]),
                                      tree("push_start",8,[])]),
                                tree("wet_weather",9,[])]),
                         tree("alternative",10,[])]),
                   tree("overheating",11,[]),
                   tree("smell",12,[]),
                   tree("vibration",13,[]),
                   tree("start_last",14,
                        [tree("bad_running",15,
                              [tree("bad_idling",16,[]),
                               tree("lack_of_power",17,[])]),
                    tree("brakes",18,[]),
                    tree("wiper",19,[]),
                    tree("lights",20,[]),
                    tree("horn",21,[])])]),
        CHOICE),
        clearwindow,
        write("Choice=",CHOICE),nl.

  test :- makewindow(5,112,112,"",5,5,20,70),
        write("You are invited to make a selection from a tree menu.\n\n"),
        write("The menu will be shown with the tree drawn in a total of\n\n"),
        write("four different attitudes.\n\n"),
        write("Each time you press F10 or ESC, or make a selection
               by pressing\n\n"),
        write("RETURN while the cursor is in a field, the menu
               will reappear\n\n"),
        write("with the tree drawn in a new attitude.\n\n"),
        write("You can move around the tree by using the cursor keys.\n\n"),
        write(".. Press the space bar to begin ......"),readkey(_),
        removewindow,fail.
  test :- show(right),fail.
  test :- show(left),fail.
  test :- show(down),fail.
  test :- show(up),fail.

goal
  repeat,test.
```

A Line Input Driver

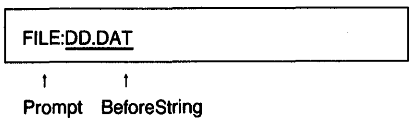
This section describes three tools, contained in LINEINP.PRO, which accept input from a user in a given screen field. In each case, a call to the corresponding tool predicate results in a window being displayed that contains the given field. Default text appears in the field. Users can press Return or F10 to accept the default text, or modify it in the usual way. The main advantage of these three tools is that they allow you to control all keystrokes—including function keys and arrow keys.

The first of the three, the tool predicate *lineinput*, is declared as follows:

```
  lineinput(ROW,COL,LEN,ATTR,ATTR,STRING,STRING,STRING)
```

It takes the form:

```
  lineinput(Row,Col,Len,Wattr,Fattr,Prompt,BeforeString,AfterString)
```



The predicate *lineinput* creates a window containing *BeforeString* at position (*Row,Col*) on the screen with attribute *Attr,* and it allows input in the field of length *Len* characters as indicated. You can either type new text or edit existing *BeforeString* text using the arrow and delete keys. Once all information is entered, the user types F10 or Return, and the modified string is returned in *AfterString*. If Esc is pressed during input, *lineinput* fails.

The two other versions of *lineinput* both take the same set of parameters. *lineinput_leave* leaves the input field on the screen after text entry has been completed; *lineinput_repeat* succeeds after each text input (and stacks a backtracking point) unless Esc is pressed, in which case it fails. Thus, several texts can be entered in the same field by using the *lineinput_repeat* ... **fail** combination. For instance,

```
   typeforever :- lineinput_repeat(3,5,40,7,7,"Type anything","",_), fail.
```

accepts input from a user and does nothing with it until he or she presses Esc.

In order to use *lineinput, lineinput_leave,* or *lineinput_repeat,* the following **database** predicates must be declared in the containing program:

```
database
   insmode
   lineinpstate(STRING,COL)
   lineinpflag
```

Using the Three *lineinput* Tool Predicates in a Program

This simple illustrative program (from XLINEINP.PRO) demonstrates all three *lineinput* tool predicates in LINEINP.PRO in turn. In each case, one or more texts are displayed, which can then be freely modified by the user. Once F10, Return, or Esc has been pressed, the resulting text input is displayed.

XLINEINP.PRO

```
include "tdoms.pro"

database
   insmode
   lineinpstate(STRING,COL)
   lineinpflag

clauses
   insmode.

include "tpreds.pro"
include "lineinp.pro"
include "status.pro"

predicates
   test

clauses
   test :-
         lineinput(10,10,45,7,7,"Which computer do you have:","IBM PC",TXT),
         write("Resulting text input: ",TXT),nl,fail.

   test :-
         changestatus("Modify each of the three texts, pressing CR after each"),
         lineinput_leave(7,10,40,7,7,"Text1: ","This is old text 1",TXT1),
         lineinput_leave(10,10,40,7,7,"Text2: ","This is old text 2",TXT2),
         lineinput_leave(13,10,40,7,7,"Text3: ","This is old text 3",TXT3),
         removewindow,removewindow,removewindow,
         write("TEXT1 = ",TXT1),nl,
         write("TEXT2 = ",TXT2),nl,
         write("TEXT3 = ",TXT3),nl,
         fail.

   test :-
         write("\nPress the space bar for a demo of lineinput_repeat"),
         readkey(_),
         clearwindow,
         changestatus("lineinput_repeat requests for new
                       lines until the ESC key is pressed."),
         lineinput_repeat(7,10,40,7,7,"Text: ","",TXT),
         shiftwindow(OLD),
         shiftwindow(2),
         write("TEXT=",TXT),nl,
         shiftwindow(OLD),
         fail.

goal
   makewindow(2,7,0,"",0,0,24,80),
   makestatus(112,"Input some new text or modify the old text"),
   test.
```

Reading a File Name

The tool predicate *readfilename* in FILENAME.PRO implements a file name input facility similar to that of Turbo Prolog's user interface. *readfilename* allows the user to type in only the file's first name; the default file type is added to that first name automatically, just as Turbo Prolog adds the .PRO extension to file names when LOADing and SAVEing programs. Alternatively, if the user just presses Return with the window empty, then a complete directory appears in a window on the screen, from which the user can select a file name by using the arrow keys and pressing Return. Again, this is the same as the Turbo Prolog system.

readfilename is declared as follows:

```
readfilename(ROW,COL,ATTR,ATTR,STRING,STRING,STRING)
```

It takes the form

```
readfilename(Row,Col,Wattr,Fattr,Extension,OldFileName,NewFileName)
```

where

- *Row* and *Col* determine the position of the input field on the screen.
- *Wattr* and *Fattr* are the window and frame attributes respectively.
- *Extension* is the STRING that is to be added to the file name if no file type is specified.
- *OldFileName* is the file name to be displayed in the window when it is first displayed. Users can edit this text and then press Return or F10 once their preferred file name is in the window.
- *NewFileName* becomes bound to the new file name with the *Extension* added automatically to the user's input if no file type was specified.

A program that uses *readfilename* must **include** the file LINEINP.PRO—in addition to TPREDS.PRO, TDOMS.PRO, and FILENAME.PRO—and contain the following **database** declarations at an appropriate point:

```
database
   insmode
   lineinpstate(STRING,COL)
   lineinpflag
```

Using *readfilename* in a Program

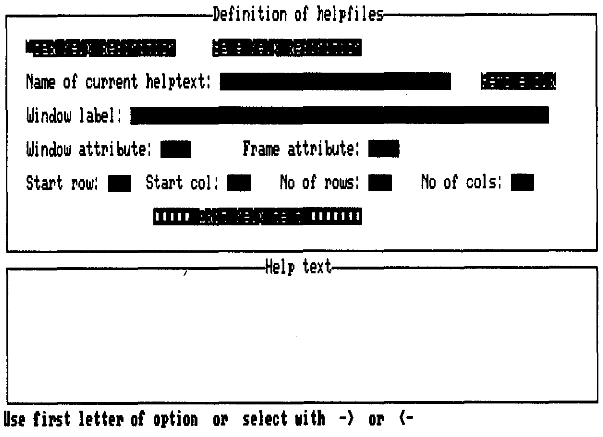
The following sample program is given in XFILENAM.PRO.

XFILENAM.PRO

```
include "tdoms.pro"

database
   insmode
   lineinpstate(STRING,COL)
   lineinpflag

include "tpreds.pro"
include "lineinp.pro"
include "filename.pro"
include "status.pro"

goal
   makewindow(2,7,7,"Instructions",0,0,24,35),disk(DIR),
   nl,write(" 1. Set the directory\n to your TOOLS dir:\n",DIR),nl,nl,
   makewindow(3,7,7,"Results",0,35,24,45),
   setdir(5,5),disk(DIR),write("Directory=",DIR),nl,nl,
   shiftwindow(2),
   write(" Now try the following in turn:"),nl,
   write(" =============================="),nl,nl,
   write("  2. Just press RETURN"),nl,
   write("  3. Give the file name \"test\""),nl,
   write("  4. Try the file name \"test.txt\""),nl,
   write("  5. Delete the old name and\n then press RETURN"),
   nl,nl,nl,nl,
   write(" - CTRL-BACKSPACE deletes the\n old file name"),
   write("\n\n\n - Use CTRL-BREAK to stop"),
   shiftwindow(3),
   makestatus(112," Give some file names and inspect the result of
            readfilename."),
   repeat,
   refreshstatus,
   readfilename(20,41,7,7,pro,"oldname.dat",NEWNAME),
   write("Filename=",NEWNAME),nl,
   fail.
```

Context-Sensitive Help

This section describes tools for the implementation of context-sensitive Help facilities. The Help texts are placed in a single Help file by running the program in HELPDEF.PRO. When that program is executed, the screen display looks like this:



```
─────────────────Definition of helpfiles──────────────────
 [illegible]          [illegible]
 Name of current helptext:                      [illegible]
 Window label:
 Window attribute:        Frame attribute:
 Start row:   Start col:    No of rows:    No of cols:
              [illegible]
─────────────────────────Help text─────────────────────────

Use first letter of option  or  select with  -> or <-
```

Figure 2.3: Screen Display for HELPDEF.PRO

Each Help text to be stored in the Help file is given a name (which can be any STRING) by typing it in the *Name of current helptext* field. Run the program, place the cursor in this field by using the arrow keys, and press Return. Type the name *help1* into the prompt window and press Return. Next, specify the string that will be used as a label for the window containing this particular Help text. Move the cursor to the beginning of the *Window label* field, and type the label `HELP FROM help1` (there's no need to press Return afterwards). Now fill in the five fields describing the window and its frame in the same way: make the window and frame attributes 7; start the window at row 4, column 10; and give it another 6 rows and 20 columns.

Now you are ready to enter the Help text itself. Move the cursor to the *Edit help text* field and press Return. Type in or edit the Help text using the standard Turbo Prolog editing commands. Enter the following text:

```
Our first example of help
Is a poem to soothe troubled users.

Roses are red
Violets are blue
```

```
Sugar is sweet
And users are too!
```

Leave the editor by pressing F10. Now save this first Help text in a file: Move the cursor to the *Save help definition* field and press Return. When prompted for a file name enter, `XHELP.DEF`.

Having done this, let's type in another Help text. Return the cursor to the *Name of current helptext* field and press Return. Give this second Help text the name `help2`. Now continue as before, labeling the containing window `MORE HELP STILL` (just type over the old text), and use the same window size attributes as last time, but start this one in row 12, column 30. The Help text for *help2* should be as follows:

```
This is help from help2.
In other words, our second
example of help.

No matter where you go,
there you are.
```

Finally, save this second help text in the *same* file, XHELP.DEF, and press Esc to terminate execution.

HELPDEF.PRO creates two files as a result of this run: XHELP.DEF (which contains details of the windows the texts are to occupy and an index to the help texts stored), and XHELP.HLP (which contains the actual texts you entered in a form that can be used by the tool predicates.)

The database tool predicate *helpcontext* implements a last in-first out stack mechanism for Help contexts, using the database tool predicate *helptext* to gain access to each Help text. When a program moves into a new context, the tool predicate *push_helpcontext* can be used to push a new Help context onto the stack. When that program leaves the current context, the tool predicate *pop_helpcontext* is used to remove the current help context and re-establish the old. If it is possible for the program to fail while in the current context, the predicate *temp_help_context* should be used instead of *push_helpcontext*—it automatically removes the Help context on backtracking.

To obtain Help in a given context, the user calls the tool predicate *help,* which is defined in the file HELP.PRO.

XHELP.PRO uses the Help file just created to illustrate these mechanisms—try it now. Having set the Help context to *help1*, the program displays Help messages from the *helptext* with that name. The parts of this text not visible in the Help window can be displayed by using the arrow keys.

To leave the Help context, press Esc. The program now enters Help context *help2*, and the appropriate Help text is displayed. Pressing Esc returns you to Help context *help1*; press Esc again to return control to the Turbo Prolog system.

Help Contexts in Tool Predicates

Many of the Toolbox predicates contain calls to make use of context-sensitive Help facilities. In the distribution versions, these Help calls have been commented out of the definitions. To reinstate them, remove the opening and ending slashes and asterisks (/* and */) from the definitions in XHELP.PRO. Users must create these Help facilities before the calls can call them.

XHELP.PRO

```
include "tdoms.pro"

domains
   HELPUNIT = h(STRING)
   HELPCONTEXT = SYMBOL
   FILEPOS = REAL
   FILE = helpfile

database
   helptext(HELPCONTEXT,ATTR,ATTR,STRING,ROW,COL,ROW,COL,FILEPOS)
   helpcontext(HELPCONTEXT)
   helpfile(STRING)

include "help.pro"

goal
   assertz(helpfile("xhelp.hlp")),
   consult("xhelp.def"),
   push_helpcontext(help1),
   help,
   push_helpcontext(help2),
   help,
   pop_helpcontext,
   help.
```

Resizing Windows

RESIZE.PRO contains the definition of the tool predicate *resizewindow*, which doesn't take any parameters. The call

```
   resizewindow
```

prompts the user to resize the current window. The arrow keys (and the Control key for larger steps) are used for resizing, just as in the Turbo Prolog system.

To move windows on most PCs, you can use the shift-arrow keys. On some PCs, you must use number keys: press 4 to move left, 2 to move down, 6 to move right, and 8 to move up.

Using *resizewindow* in a Program

XRESIZE.PRO demonstrates the use of *resizewindow*. Notice that you are expected to provide defining clauses for the predicate *writescr,* which is used to refresh the screen during window resizing. Its declaration is contained in RESIZE.PRO. Note also that the definition of *resizewindow* uses predicates defined in STATUS.PRO, so STATUS.PRO must be **included** in any program that uses *resizewindow.*

XRESIZE.PRO

```
include "tdoms.pro"

database
   windowsize(ROW,COL)

predicates
   writescr

include "tpreds.pro"
include "status.pro"
include "resize.pro"


clauses
   writescr :- keypressed,!.
   writescr :-
         write("The resizing tool predicate allows a window to be resized\n"),
         write("in the same way as with the Turbo Prolog system.\n\n"),
         fail.
   writescr :- keypressed,!.
   writescr :-
         attribute(112),
         write("The main advantage of the tool version is that
                you can modify\n"),
         write("the resize predicate to suit your own needs.\n\n"),
         attribute(7).

goal
   makewindow(1,7,0,"",0,0,24,80),
   makewindow(2,7,7,"test",5,5,14,70),
   resizewindow.
```

A Mixed Bag of BIOS Calls

This section describes a few functions that are easily implemented with calls to the BIOS standard predicate. The definitions of the various BIOS calling tool predicates are collected together in the file BIOS.PRO. Some of the tool predicates are implemented as described in the *Turbo Prolog Owner's Handbook* (see "Low-Level Support," Chapter 11). These are

- *dosver(REAL),* which returns the DOS version number;
- *diskspace(INTEGER,REAL,REAL),* which, for a given disk, returns the total available disk storage space and the number of bytes currently free;
- *mkdir(STRING),* which creates a new subdirectory; and
- *rmdir(STRING),* which removes a subdirectory.

Four others are provided in the Toolbox: *getverify* and *setverify* for setting and reading the BIOS *verify* switch; *border* for changing the color of the screen border; and *findmatch,* which can be used to obtain directory information from BIOS.

Setting and Reading the Verify Switch

When the BIOS verify switch is ON, every DOS write operation is followed by a read operation to ensure that the data has been stored correctly. *getverify* returns the state of the verify switch, and *setverify* sets the verify switch. They are both called with an INTEGER parameter. For example, if a call

```
getverify(Switch)
```

binds *Switch* to *0,* then verify is OFF; whereas if *Switch* is bound to *1,* then verify is ON.

Similarly,

```
setverify(1)
```

turns verify ON and

```
setverify(0)
```

turns verify OFF.

Changing the Color of the Screen Border

The tool predicate *border* is used to change the color of the border of the screen when the screen is working in the 25*80 alphanumeric mode. For example,

```
border(15)
```

changes the color of the border to white.

Reading File Names from the Directory

It is sometimes necessary to find files on a certain disk, disk drive, or directory, or to obtain the size or creation date of a file. The tool predicate *findmatch* is very useful in this connection, since it allows the program to search a directory for file names that match a given string mask. All available matching-file information is returned in *findmatch*'s parameters. *findmatch* takes a total of ten parameters, the first two being input parameters and the remainder being output parameters. Thus, in a call

```
findmatch(SearchFileSpec,SearchAttr,MatchFileName,MFAttr,
        MFHour,MFMin,MFYear,MFMonth,MFDay,MFSize)
```

SearchFileSpec is a normal DOS file specification (for example, of the form C:*.*), and *SearchAttr* is a bitmask indicating the bounds of the directory search as follows:

0	ordinary files
1	read-only files
2	hidden files
4	system files
8	volume label
16	subdirectory
32	archive files (used by BACKUP and RESTORE)

So that

```
findmatch("B:\\SPECIAL\\*.PRO",3,MatchFileName,MFAttr,
        MFHour,MFMin,MFYear,MFMonth,MFDay,MFSize)
```

binds *MatchFileName* to the name of a file in the SPECIAL directory on Drive B:, which is either a hidden file or a read-only file; *MFAttr* to a value indicating its attribute mask according to the above table; *MFHour, MFMin, MFYear, MFMonth,* and *MFDay* to the creation time and date of that file; and *MFSize* to the size of that file.

C H A P T E R

3

Screen-Layout Tools

The Turbo Prolog Toolbox provides tool predicates for designing and implementing screen layouts easily. In this chapter, you'll learn how to define a screen layout and how to edit, save, and load the definition. You'll associate values and actions with fields—and specify *no input* fields. A few sample programs follow, then more advanced techniques are discussed: specifying new keys and screen-definition types, producing interchangeable screen layouts, and creating new screen definitions from old. Finally, formatting and printing reports are explained.

If you follow through the examples and type in the instructions shown, you'll find it easier to understand the various concepts behind screen-layout definitions.

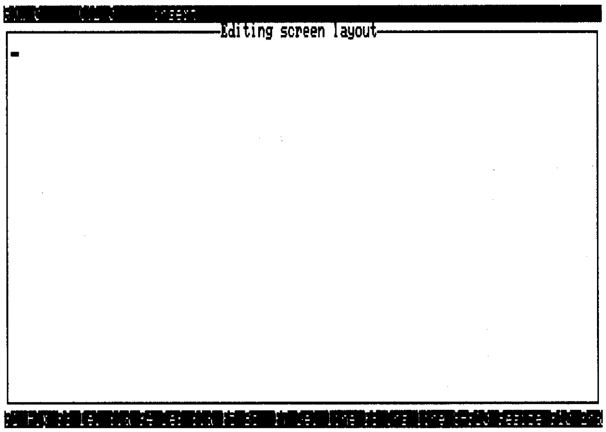
Many of these screen tools are not restricted to the row and column dimensions of the PC screen; they allow you to work with a virtual screen. So you can offer users any size of screen—though, of course, only one monitor-sized screenful of a larger window can be viewed at a time. Moreover, the screen-layout tools are designed so that you can work with a PC-sized screen first, then move to virtual-screen capabilities only when it becomes necessary: The real-screen tools and the virtual-screen tools are entirely compatible.

Whether you're working with real screens or virtual screens, *screen-layout definitions* are created by running the program in SCRDEF.PRO. These layout definitions are then used by the screen-handling tool predicate *scrhnd*. The definition of *scrhnd* for real screens is contained in SCRHND.PRO; for virtual screens, that definition is in VSCRHND.PRO. These definitions are entirely compatible. In fact, you can use VSCRHND.PRO even when working with real displays, although the total program code size will be greater.

scrhnd enables you to associate *actions* and a *value* with each data field specified in a layout definition. For example, having defined two fields in which prices are entered, the *action* associated with each of these two fields may be to pop up a menu from which items with specified prices can be selected, and to add the price of the item chosen to a running total. This running total can then be used to determine the *value* to be displayed in a third field that is reserved for the total cost of the two items.

Basic Screen-Layout Definition

Run the program SCRDEF.PRO and select the Define Screen Layout option from its main menu. The screen should now look like Figure 3.1.

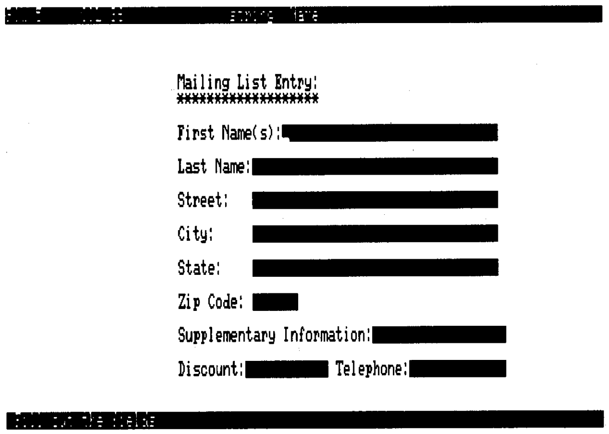


```
Editing screen layout
```

Figure 3.1: The Define Screen Layout Option

To specify fields for text display or data entry with SCRDEF.PRO, move the cursor to the desired position and use the Function keys to define a field.

Let's work through an example that will result in the following screen layout.



```
Mailing List Entry:
******************

First Name(s):

Last Name:

Street:

City:

State:

Zip Code:

Supplementary Information:

Discount:            Telephone:
```

Figure 3.2: Screen Layout for Mailing List Entries

To define this screen layout, first move the cursor to row 2, column 22 (or 2,22) of the screen (the top left-hand corner of the screen is row 0, column 0). Type in the phrase `Mailing List Entry`. Then move the cursor to position (3,22) and type in `****************`. Next, move the cursor to (5,22) and enter `First Name(s):`. You have defined three *fixed fields* of text.

Next, move the cursor to position (5,36), where the first *data field* should start. Define this position to be the start of a data field by pressing F4. (The status line at the bottom of the screen shows the effects of the various function keys.) SCRDEF.PRO then prompts for the name by which the defined field will be known within a Prolog program: Let's call it *FirstNames*. Type this in the prompt window and press Return. You are prompted for the type of the field. Since this must be a string field, select the String option from the menu and press Return. Now define the length of the field by pressing the right arrow key until a field of the correct size is shaded on the screen. Press the right arrow key twenty-three times and observe how the data field is shaded. Now press Return, and the definition is complete.

Since the cursor is still in a field, the name of that field and its type (if it is a data field) are displayed at the top of the screen. Move the cursor to position (7,22). Note that when you move the cursor from the data field just defined, the relevant information at the top of the screen is deleted.

The remaining fields in Figure 3 are defined similarly; you should now complete them. When you have defined all fields, press F10. To save this layout definition in a file, select the Save Screen Layout option from the menu. When prompted for a file name, save this definition under the name XMAIL.SCR.

The contents of XMAIL.SCR constitute a screen definition file. Such a file contains clauses for three **database** predicates corresponding to the following declarations:

```
database
  field(FieldName,Type,Row,Column,Length)
  textfield(Row,Column,Length,FieldString)
  windowsize(Height,Width)
```

The **database** predicate *windowsize* uses *Height* and *Width* to determine the size of the window through which the user can view the real or virtual screen. Clauses for *textfield* specify the text fields in the (virtual) screen layout, with each text field specified by its position (*Row,Column*), the *Length* of the field, and its contents (*FieldString*). Clauses for the **database** predicate *field* specify the data fields: *FieldName* gives the name of a field, *Row* and *Column* specify the starting position of that field within the (virtual) screen window, *Length* names the length of that field, and *Type* specifies which type of data this field can accept.

In this example, SCRDEF.PRO generates the following text-file description of the screen layout (to see this description, select the Edit Screen Definition File option of SCRDEF.PRO and enter the file XMAIL.SCR).

```
field("FirstNames",str,5,36,23)
field("LastName",str,7,33,26)
field("Street",str,9,33,26)
field("City",str,11,33,26)
field("State",str,13,33,26)
field("ZipCode",str,15,33,8)
field("SuppInfo",str,17,49,10)
field("Discount",real,19,31,8)
field("Tel",str,19,50,14)
txtfield(2,22,19,"Mailing list entry:")
txtfield(3,22,19,"*******************")
txtfield(5,22,14,"First Name(s):")
txtfield(7,22,11,"Last Name: ")
txtfield(9,22,7,"Street:")
txtfield(11,22,11,"City:      ")
txtfield(13,22,11,"State:     ")
txtfield(15,22,9,"ZipCode:")
txtfield(17,22,27,"Supplementary information: ")
txtfield(19,22,9,"Discount:")
txtfield(19,39,11," Telephone:")
windowsize(20,77)
```

This example layout design is specified. Normally, however, you would run SCRDEF.PRO with only the most basic of layout specifications and experiment with various layouts until a satisfactory layout is achieved.

SCRDEF.PRO provides tools to help you play around with different layouts, including: deletion of a text or data field previously defined (press F3), insertion of a virtual screen line (F8), and the ability to resize the containing window (Shift-F10). The section "Facilities in SCRDEF.PRO" shows you how to use these facilities and describes how it is possible for screen layouts to contain boxes (drawn with the cursor keys). In the next section, however, you'll see how the screen handler, *scrhnd*, uses the screen definition just created.

Basic Use of the Screen Handler

The program XMAIL.PRO makes use of a screen-layout definition and calls the tool predicate *scrhnd* from either SCRHND.PRO or VSCRHND.PRO. Since the definitions are compatible, the following example uses SCRHND.PRO.

The sequence

```
consult("xmail.scr"),
createwindow(TopLineSwitch),
scrhnd(TopLineSwitch, KeyUsedForReturn)
```

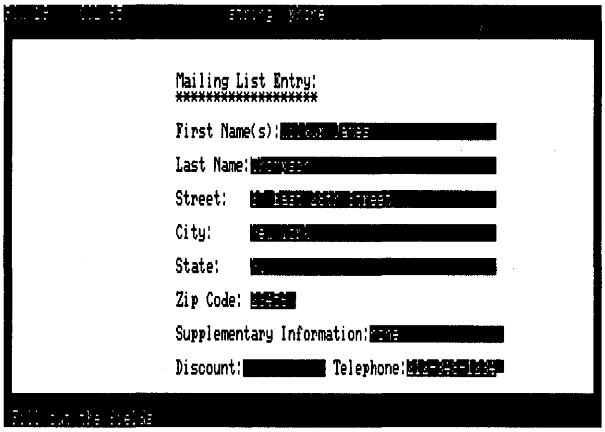
contains *scrhnd*, which activates the screen-layout defined in XMAIL.SCR.

The second line, *createwindow*, is a tool predicate that creates a window in which the screen-layout specification obtained by *consult*ing MAILDEF.SCR is displayed.

scrhnd takes two parameters and uses the values *consult*ed from XMAIL.SCR to form a screen display. The application's user then types information into the appropriate fields in this screen—a name, address, and telephone number, for example. Indeed, the user can fill in the entries for each field and edit them in various order until he or she terminates input by pressing F10 or Esc.

KeyUsedForReturn, the parameter in the third line of the example sequence, is bound to the code for the key that terminates the screen handler. If *TopLineSwitch* is bound to ON, a line is displayed at the top of the screen giving the name of the field that currently contains the cursor. No such line is displayed if *TopLineSwitch* is bound to OFF.

Suppose that the user makes the entries shown in Figure 3.3.



```
Mailing List Entry:
*******************

First Name(s): [illegible]

Last Name: [illegible]

Street: [illegible]

City: [illegible]

State: [illegible]

Zip Code: [illegible]

Supplementary Information: [illegible]

Discount:            Telephone: [illegible]
```

Figure 3.3: Sample Mailing-List Entries

On return from *scrhnd,* entries are automatically made in the database for the predicate *value* declared by

```
database
   value(FieldName,String)
```

as follows:

```
   value("FirstNames","Wilbur James")
   value("LastName","Thompson")
   value("Street","37 East 28th Street")
   value("City","New York")
   value("State","NY")
   value("Tel","212-348-1234")
   value("ZipCode","23456")
   value("Discount","28")
```

The following sample application (XMAIL.PRO) allows users to enter information using the screen layout defined earlier and then displays an address label constructed from the data entered. The program assumes that the example screen layout in the preceding paragraphs has been saved in a file called XMAIL.SCR. All the domain and (**database**) predicate definitions necessary for using *scrhnd* are contained in the **include** file HNDBASIS.PRO.

XMAIL.PRO

```
include "handbasis.pro"

goal
   consult("xmail.scr"),
   createwindow(on),
   scrhnd(on,_),
   clearwindow,
   value("FirstNames",FName),
   value("LastName",LName),
   value("Street",Street),
   value("City",City),
   value("State",State),
   value("Tel",Tel),
   value("ZipCode",Zip),
   value("Discount",Discount),nl,nl,TAB
   write("+------------------------------------------------------------"),nl,
   write(FName," ",LName,"\n",Street,"\n",City,"\n",State,"\n"),nl,
   write("Your reference number will be your telephone number:",Tel),nl,
   write("You have been granted ",Discount,"% discount."),nl,
   write("+------------------------------------------------------------").
```

Facilities in SCRDEF.PRO

You can use the program SCRDEF.PRO to generate screen-layout definitions along the lines of the earlier examples. When SCRDEF is executed, the following menu appears.

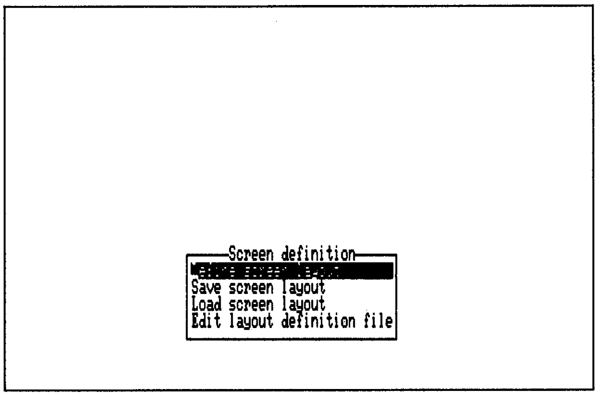


Figure 3.4: Menu Displayed by SCRDEF.PRO

You can then do one of the following: **D**efine a screen layout, **S**ave the current screen-layout definition in a file, **L**oad a previously saved screen-layout definition, or **E**dit (or browse) through a file containing a screen-layout definition. Pressing the first letter of each menu item (shown here in boldface type) calls up that command. In the following sections, each of these commands are described in more detail.

Define Screen Layout

The Define command enables the screen-layout editor so that you can define a new screen layout or modify an existing one. Editing takes place in a window that initially has 21 rows and 78 columns. This window is your view on a virtual screen having up to 32,768 rows and 32,768 columns. You move around this virtual screen using the usual Turbo Prolog editor cursor-movement commands (listed in Chapter 12 of the *Turbo Prolog Owner's Handbook.*

During the Define command execution, the top of the screen holds a status line describing the current cursor position. If the cursor is positioned in a defined field, the status line also contains a description of that field, giving the field name and type. During Edit, a Help line at the bottom of the screen contains a list of those function keys having special meaning during layout definition:

```
F1:Hlp F3:Del fld F4:Def fld F5:Box F7:Del line F8:Ins line S-F10:Resize F10:End
```

The functions provided by these keys are as follows:

Help facilities F1
Pressing F1 causes a window containing Help text to pop up; you can either browse through the text or search for a specific text using the Turbo Prolog editor's Search command. Pressing Esc or F10 returns control to the screen-layout editor.

Inserting fixed fields
Pressing any key other than a cursor-movement or function key immediately defines a fixed field containing that character and any text typed to its right.

Deleting a field F3
If the cursor is currently positioned anywhere in a defined field, pressing F3 deletes that field from the screen-layout definition.

Defining data fields F4
F4 is used to define a data field. When pressed, you are prompted for a name by which that field will be known. Next, the type of field required must be specified as INTEGER, REAL, or STRING. (This type-checking facility is easy to expand and/or modify as described later in "Adding Your Own Screen-Definition Types.") Finally, you must specify the length of the field, using the arrow keys. Terminate the field definition by pressing Return.

Enter box-drawing mode F5
Pressing F5 switches you into box-drawing mode, where the cursor keys are used to draw straight line segments. Screen layouts can thus incorporate boxes, which can be used to group related fields or to organize the layout. Boxes can be drawn anywhere on the virtual screen; also, they can overlap. Press F10 to leave box-drawing mode and return the cursor keys to their normal function. The box-drawing facility is most effectively used by swapping between normal and box mode and using the space bar to delete drawing errors as necessary. Extended ASCII graphics characters are stored in fixed fields to represent the line segments that make up the boxes.

Delete a line F7
To remove a line on the screen, press F7.

Insert a line F8
To insert a blank line on the screen, press F8.

Resize the window Shift-F10
To resize the screen-layout editor window, hold Shift down and press F10. A new Help line appears at the bottom of the screen that specifies which keys may be used to resize the window (which, by default, is as large as it can be).

Return to the main menu F10, Esc
Press F10 or Esc.

Save Screen Layout

This command saves a screen-layout definition in a disk file. After the `File Name:` prompt, the user can press Return to view a box menu of .SCR files and check which file names have already been used. Pressing Esc then takes the user back a step to the `File Name:` prompt.

Load Screen Layout

This command loads a previously saved screen layout from a disk file. Pressing Return in response to the `File Name:` prompt produces a box menu of .SCR files, from which you can select a file to work with. When the screen-definition file is loaded, SCRDEF.PRO checks that

- the clauses conform to the syntax for the appropriate database predicate,
- no two fields have the same name,
- no two fields overlap,
- any string in a text field is not longer than the length of that text field.

If an error is detected, the main Turbo Prolog editor is entered, and the cursor is positioned at the error. Once the error has been corrected and the editor exited by pressing F10, the Load operation is rerun from the beginning.

Edit Layout Definition File

This command enables the user to edit or browse through a screen-definition file. Be careful when editing defining clauses here; doing so can lead to inconsistencies.

Associating Values with Fields

In order to demonstrate how values are associated with fields in a screen-layout definition, let's work through a small example. This time, it's a program designed to simulate a simple adding machine. The user types in two numbers, and a field shows the total of the entries (if any) in the two fields. When the program is executed, the display looks like Figure 3.5.

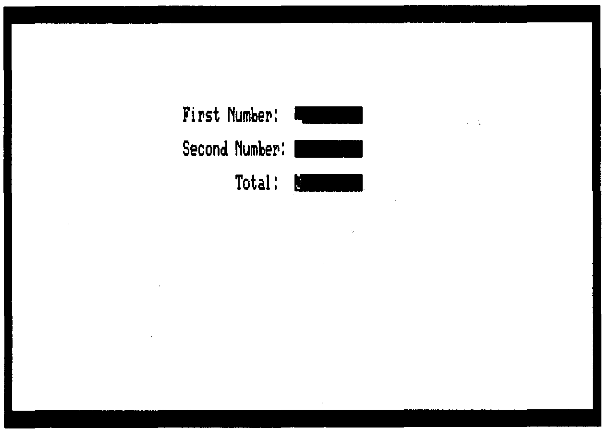


Figure 3.5: Screen Layout for the Adding Machine Simulator

The first step is to define the screen layout using SCRDEF.PRO. Let's assume that you've already done this, and the screen-layout definition is in the file ADDERDEF.SCR. Assume further that the names of the fields are *FirstNo*, *SecondNo*, and *Total* and that they are all of REAL type. The contents of ADDERDEF.SCR thus resemble the following:

```
field("FirstNo",real,5,38,9)
field("SecondNo",real,7,38,9)
field("Total",real,10,38,9)
txtfield(5,23,13,"First Number:")
txtfield(7,23,14,"Second Number:")
txtfield(10,30,6,"Total:")
windowsize(21,77)
```

When this screen-definition file is used in conjunction with *scrhnd,* the user is presented with the specified screen layout and should type in strings corresponding to valid real numbers for the *FirstNo* and *SecondNo* fields. *scrhnd* records these values as strings in the database for the *value* predicate. Thus, if the user types 23 and 76 respectively, the corresponding database entries are

```
value("FirstNo","23").
value("SecondNo","76").
```

In order to associate a value with the *Total* field, which depends on the *FirstNo* and *SecondNo* values entered by the user, use the tool predicate *field_value.* It takes two parameters, the name of a field and the value to be displayed in that field. Thus,

```
field_value("Total",T) :-
     value("FirstNo",N1),
     str_real(N1,Num1),
     value("SecondNo",N2),
     str_real(N2,Num2),!,
     Tnum = Num1+Num2,
     str_real(T,Tnum).

field_value("Total",N1) :- value("FirstNo",N1),!.

field_value("Total",N2) :- value("SecondNo",N2),!.

field_value("Total","0").
```

defines the value to be displayed in the *Total* field as: the sum of *FirstNo* and *SecondNo,* if both have been entered; the value of one, if that is all that the user entered; or zero, if the user hasn't made any entries.

The final step is to update a copy of HNDBASIS.PRO by editing it to **include** the above clauses for *field_value.* HNDBASIS.PRO provides the basic framework for programs that use the screen-handling tools. It contains all the necessary database predicate and domain definitions, and a reminder that, in general, a program using *scrhnd* should define clauses for the *field_action, field_value,* and *noinput* tool predicates.

HNDBASIS.PRO

```
/*************************************************************************
A minimal structure for a program that uses the screen handlers.
*************************************************************************/

/*************************************************************************
                              Domains
**************************************************************************/

include "tdoms.pro"

domains
   FNAME = SYMBOL
   TYPE = int(); str(); real()

/*************************************************************************
                        Database Predicates
*************************************************************************/

database

/* Database declarations used in SCRHND */
   insmode                                          /* Global insertmode */
   actfield(FNAME)                                      /* Actual field */
   screen(SYMBOL,DBASEDOM)                    /* Saving different screens */
   value(FNAME,STRING)                              /* Value of a field */
   field(FNAME,TYPE,ROW,COL,LEN)                   /* Screen definition */
   txtfield(ROW,COL,LEN,STRING)
   windsize(ROW,COL).
   notopline

/* Database predicates used by VSCRHND */
   windstart(ROW,COL)
   mycursord(ROW,COL)

/* Database declarations used in LINEINP */
   lineinpstate(STRING,COL)
   lineinpflag

/*************************************************************************
                           Include tools
*************************************************************************/

include "tpreds.pro"
include "menu.pro"
include "status.pro"
include "lineinp.pro"
include "scrhnd.pro"

clauses
```

```
/*************************************************************************
                           field_action
*************************************************************************/

   field_action(_) :- fail.

/*************************************************************************
                           field_value
*************************************************************************/

   field_value(FNAME,VAL) :- value(FNAME,VAL),!.

/*************************************************************************
                            noinput
*************************************************************************/

   noinput(_) :- fail.
```

Note that the default value for a field is defined as the value the user types by the clause

```
   field_value(FNAME,VAL) :- value(FNAME,VAL),!.
```

Now, copy HNDBASIS.PRO into the editor and move your cursor to the first line after the *field_value* comment line. Insert the clauses worked out a few pages earlier for the *Total* field's value—all ten lines of code. Then, at the very end of the program, type in these lines:

```
goal
   consult("ADDERDEF.SCR"),
   createwindow(off),
   scrhnd(off,_).
```

Save the modified file under the name MYADDER.PRO

Compile and execute the result. Experiment with the adding machine produced, using the arrow keys to select the two input fields. Notice what happens when you type a new value over an existing one, type in non-numeric values, or press Del.

Associating Actions with Fields

The tool predicate *field_action* takes one parameter—a field name—and allows one or more actions to be carried out if Return is pressed while the cursor is in that field. Let's work on the example you just created: You'll arrange things so that whenever Return is pressed in the *Total* field, the current value of the *Total* field is copied into a new field, the *Memory* field, which functions like the Memory button on some calculators.

First, redefine the screen layout as shown in Figure 3.6.

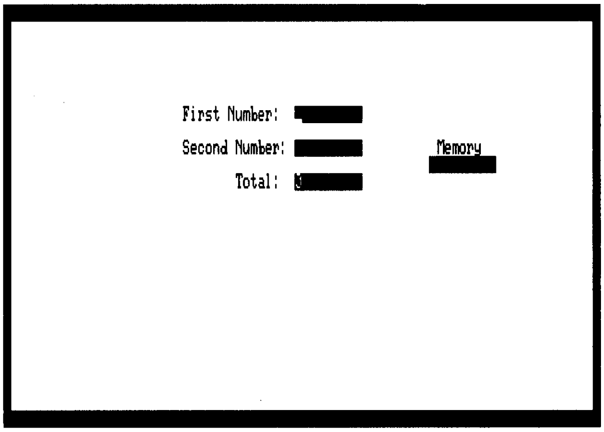


Figure 3.6: The Adding Machine with Memory

Save results in a screen-definition file, XMADDER.SCR, whose contents should be similar to the following:

```
field("FirstNo",real,5,38,9)
field("SecondNo",real,7,38,9)
field("Memory",real,8,56,9)
field("Total",real,9,38,9)
txtfield(9,30,6,"Total:")
txtfield(5,23,13,"First Number:")
txtfield(7,23,14,"Second Number:")
txtfield(7,57,6,"Memory")
windowsize(21,77)
```

Next, take MYADDER.PRO, which is actually HNDBASIS.PRO with the following altered *field_value* clauses:

```
field_value("Total",T) :-
      value("FirstNo",N1),
      str_real(N1,Num1),
      value("SecondNo",N2),
      str_real(N2,Num2),!,
      Tnum = Num1+Num2,
      str_real(T,Tnum).

field_value("Total",N1) :-
      value("FirstNo",N1),!.

field_value("Total",N2) :-
      value("SecondNo",N2),!.

field_value("Total","0").

field_value(FNAME,VAL) :-
      value(FNAME,VAL),!.
```

Change the field action for *Total* from the default defined by

```
field_action(_) :- !, fail.
```

to

```
field_action("Total") :-
      retract(value("Memory",_)),fail.

field_action("Total") :-
      field_value("Total",T),
      assertz(value("Memory",T)).
```

When Return is pressed in the *Total* field, therefore, the current value of *Total* is asserted for *Memory*.

Now, change the goal so that XMADDER.SCR is consulted rather than ADDERDEF.SCR, and you're done.

The following program, XMADDER.PRO, shows the completed code. Try modifying it so that when Return is pressed in either *FirstNo* or *SecondNo*, the current value of *Memory* becomes the value of that field (while the value of *Memory* remains unaltered).

XMADDER.PRO

```
check_determ code=2000

include "tdoms.pro"

domains
   FNAME = SYMBOL
   TYPE = int(); str(); real()

database
/* Database declarations used in SCRHND */
   insmode                                              /* Global insertmode */
   actfield(FNAME)                                      /* Actual field */
   screen(SYMBOL,DBASEDOM)                    /* Saving different screens */
   value(FNAME,STRING)                                  /* Value of a field */
   field(FNAME,TYPE,ROW,COL,LEN)                      /* Screen definition */
   txtfield(ROW,COL,LEN,STRING)
   windowsize(ROW,COL).
   notopline

/* Database declarations used in VSCRHND */
   windowstart(ROW,COL)
   mycursord(ROW,COL)

/* Database declarations used in LINEINP */
   lineinpstate(STRING,COL)
   lineinpflag

include "tpreds.pro"
include "menu.pro"
include "status.pro"
include "lineinp.pro"
include "scrhnd.pro"

goal
   consult("xmadder.scr"),
   createwindow(off),
   makestatus(112," Type some numbers"),
   scrhnd(off,_).

clauses

/*************************************************************************
                           field_action
*************************************************************************/

   field_action("Total") :-
        retract(value("Memory",_)),fail.

   field_action("Total") :-
        field_value("Total",T),
        assertz(value("Memory",T)).
```

```
/**********************************************************************
                               field_value
**********************************************************************/

   field_value("Total",T) :-
        value("FirstNo",N1),
        str_real(N1,Num1),
        value("SecondNo",N2),
        str_real(N2,Num2),!,
        Tnum = Num1+Num2,
        str_real(T,Tnum).

   field_value("Total",N1) :-
        value("FirstNo",N1),!.

   field_value("Total",N2) :-
        value("SecondNo",N2),!.

   field_value("Total","0") :-!.

   field_value(FNAME,VAL) :-
        value(FNAME,VAL),!.

/**********************************************************************
                               noinput
**********************************************************************/

   noinput(_) :- fail.
```

No-Input Fields

You may have spotted one slight problem with the adding-machine-plus-memory program: The user can type values into the *Memory* field. Fortunately, any such user-initiated entry is overridden when the program makes a proper entry in that field. Nevertheless, it's still an untidy feature. And the same failing applies to the *Total* field, although the effect is less noticeable.

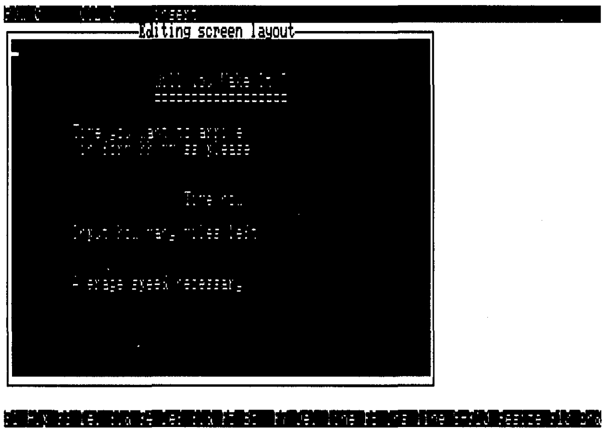
Thanks to the *noinput* predicate, blocking users from entering text into specific fields is easy to arrange. In this case, just specify the *Memory* and *Total* fields as no-input fields by replacing the default defining clause for *noinput* (near the end of the program) in HNDBASIS as follows:

```
   noinput("Memory").
   noinput("Total").
```

Make these alterations, then verify that you can't enter information in the *Memory* and *Total* fields.

Computing Travel Time

Suppose you need to catch a plane and want to know how fast you'll have to drive to get to the airport on time. Naturally, you have with you your computer, the Turbo Prolog system and Toolbox, and a source of power. You call up the following program, XTRAVEL.PRO, which looks at the desired arrival time and the distance left to go, and works out the average speed necessary to meet the deadline. It uses the screen-layout definition) in XTRAVEL.SCR, which the screen handler displays in the format shown in Figure 3.7.



```
Editing screen layout

[illegible]
=================

Time you want to arrive
[illegible] please

Time now

Input how many miles left

Average speed necessary
```

Figure 3.7: The XTRAVEL Screen Layout

Going from top to bottom, the useful data field names are *arrivetime, timenow, miles,* and *speed.* There is also a field called *flip,* which displays a speedometer dial (and demonstrates just how powerful the Toolbox screen-handling facilities are).

Another useful feature is that the *timenow* field is continually updated, like a real time clock. You accomplish this by defining the field value for *timenow* as follows:

```
field_value(timenow,TIME) :-
    time(Ho,M,S,H),
    T = S+((H+50) div 100),
    str_int(Seconds,T),std(Seconds,SecondsSt),
    str_int(Minutes,M),std(Minutes,MinutesSt),
    str_int(Hour,Ho),
          std(Hour,HourSt),
          concat(HourSt,":",S1),concat(S1,MinutesSt,S2),concat(S2,":",S3),
          concat(S3,SecondsSt,Time).
```

Here, the predicate *std* ensures that the time is always displayed in the standard hh:mm:ss format. The standard predicate *time* accesses the system clock.

The user can reset the clock whenever Return is pressed with the cursor in the *timenow* field. Hence, a field action for *timenow* is defined as follows:

```
   field_action(timenow) :-
         makewindow(1,7,7,"Set new time",5,5,8,40),
         write("Hours:\t"),readint(H),nl,
         write("Minutes:\t"),readint(M),nl,
         write("Seconds:\t"),readint(S),
         time(H,M,S,0),fail.

   field_action(timenow) :-
         removewindow.
```

To determine the necessary speed and display it in the *speed* field, convert the strings into numbers:

```
   field_value(speed,NecSpeed) :-
         field_value(timenow,TimeStr),
         value(arrivetime,ArriveStr),
         timediff(ArriveStr,TimeStr,TimeSecs),
         value(miles,MileStr),!,
         str_real(MileStr,NoOfMiles),
         RealSpeed = NoOfMiles/(TimeSecs/3600),
         str_real(NecSpeed,RealSpeed).

   field_value(Fn,X) :-
         value(Fn,X),!.
         {/* Values from normal user input*/}
```

As you can see, *timediff(ArriveStr,TimeStr,TimeSecs)* returns *TimeSecs* as the difference, in seconds, between *ArriveStr* and *TimeStr*, which are time periods represented as strings in the form hh:mm:ss.

The no-input section is easy:

```
   noinput(speed).
   noinput(timenow).
```

To complete the basic program, give it a goal and define the predicates used to specify the field values and actions:

```
domains
   ILIST = INTEGER*
```

```
predicates
   timesplit(STRING,INTEGER,INTEGER,INTEGER)
   timediff(STRING,STRING,REAL)
   str_intgl(STRING,ILIST)

goal
   consult("xtravel.scr"),
   createwindow(off),
   scrhnd(off,_).

clauses
   str_intgl("",[]) :- !.
   str_intgl(STR,[H|T]) :-
         fronttoken(STR,INTSTR,REST),
         str_int(INTSTR,H),!,
         str_intgl(REST,T).
         str_intgl(STR,T) :-
         fronttoken(STR,_,REST),
         str_intgl(REST,T).

   timesplit(TimeStr,H,M,S) :-
         str_intgl(Timestr,Intglist),
         Intglist = [H,M,S].

   timediff(TimeStr1,TimeStr2,DiffSecs) :-
         timesplit(TimeStr1,H1,M1,S1),
         timesplit(TimeStr2,H2,M2,S2),
         TotSecs1 = H1*3600.0+M1*60+S1,
         TotSecs2 = H2*3600.0+M2*60+S2,
         DiffSecs = TotSecs1-TotSecs2.
```

All that remains is to complete the speedometer. To do so, add the **database** predicate declaration

```
   flip(STRING)
```

to the existing ones. This defines a field value for the field *flip* as follows:

```
   field_value(flip,STR2) :- !,
         retract(flip(STR)),
         frontchar(STR,CH,STR1),
         str_char(CHS,CH),
         concat(STR1,CHS,STR2),
         assertz(flip(STR2)),!.
```

You also need to add the clause

```
   flip("~~~~~~~~~~~~~~~~                    ").
```

Here's the complete program listing.

```
check_determ code=2000

include "tdoms.pro"

domains
   FNAME = SYMBOL
   TYPE = int(); str(); real()

database
/* Database declarations used in SCRHND */
   insmodeT                                        /* Global insertmode */
   actfield(FNAME)                                 /* Actual field */
   screen(SYMBOL,DBASEDOM)                         /* Saving different screens */
   value(FNAME,STRING)                             /* Value of a field */
   field(FNAME,TYPE,ROW,COL,LEN)                   /* Screen definition */
   txtfield(ROW,COL,LEN,STRING)
   windowsize(ROW,COL).
   notopline

/* Database predicates used by VSCRHND */
   windstart(ROW,COL)
   mycursord(ROW,COL)

/* Database declarations used in LINEINP */
   lineinpstate(STRING,COL)
   lineinpflag

   flip(STRING)

include "tpreds.pro"
include "status.pro"
include "lineinp.pro"
include "scrhnd.pro"

predicates
   timesplit(STRING,INTEGER,INTEGER,INTEGER)
   timediff(STRING,STRING,REAL)
   std(STRING,STRING)

goal
   consult("xtravel.scr"), createwindow(off), scrhnd(off,_).

clauses

/************************************************************************
                         field_action
************************************************************************/

   field_action(timenow) :-
         makewindow(1,7,7,"Set new time",5,5,8,40),
         write("Hours:\t"),readint(H),nl,
         write("Minutes:\t"),readint(M),nl,
        write("Seconds:\t"),readint(S),
        time(H,M,S,0),fail.

   field_action(timenow) :-
        removewindow.

   field_action(speed) :-
        makewindow(1,7,7,"ERROR Window",5,5,8,40),
        write("\n\n This is not an input field"),
        sound(100,300),
        Readchar(_),
        removewindow.

/***********************************************************************
                              field_value
***********************************************************************/

   field_value(timenow,TIME) :- !,
        time(Ho,M,S,H),
        T = S+((H+50) div 100),
        str_int(Seconds,T),std(Seconds,SecondsSt),
        str_int(Minutes,M),std(Minutes,MinutesSt),
        str_int(Hour,Ho),
        std(Hour,HourSt),
        concat(HourSt,":",S1),concat(S1,MinutesSt,S2),concat(S2,":",S3),
        concat(S3,SecondsSt,Time).

   field_value(speed,NecSpeed) :- !,
        field_value(timenow,TimeStr),
        value(arrivetime,ArriveStr),
        timediff(ArriveStr,TimeStr,TimeSecs),
        value(miles,MileStr),!,
        str_real(MileStr,NoOfMiles),
        RealSpeed = NoOfMiles/(TimeSecs/3600),
        str_real(NecSpeed,RealSpeed).

   field_value(flip,STR2) :- !,
        retract(flip(STR)),
        frontchar(STR,CH,STR1),
        str_char(CHS,CH),
        concat(STR1,CHS,STR2),
        assertz(flip(STR2)),!.

/* Values from normal user input */
   field_value(Fn,X) :- value(Fn,X),!.

   flip("~~~~~~~~~~~~~~~~                        ").

/***********************************************************************
                              noinput
***********************************************************************/

   noinput(speed).
   noinput(timenow).
   noinput(flip).

/***********************************************************************
                       Intermediate predicates
***********************************************************************/
```

```
domains
   ILIST = INTEGER*

predicates
   str_intgl(STRING,ILIST)

clauses
   str_intgl("",[]) :- !.
   str_intgl(STR,[H|T]) :-
         fronttoken(STR,INTSTR,REST),
         str_int(INTSTR,H),!,
         str_intgl(REST,T).
         str_intgl(STR,T) :-
         fronttoken(STR,_,REST),
         str_intgl(REST,T).

   timesplit(TimeStr,H,M,S) :-
         str_intgl(Timestr,Intglist),
         Intglist = [H,M,S].

   timediff(TimeStr1,TimeStr2,DiffSecs) :-
         timesplit(TimeStr1,H1,M1,S1),
         timesplit(TimeStr2,H2,M2,S2),
         TotSecs1 = H1*3600.0+M1*60+S1,
         TotSecs2 = H2*3600+M2*60+S2,
         DiffSecs = TotSecs1-TotSecs2.

   std(H,HS) :- str_len(H,L),L<2,!,concat("0",H,HS).
   std(H,H).
```

Three Sample Programs

Following are three programs that show you different ways to use screen handlers. One sorts through your database and pulls out entries that match your specifications. The second helps you record sales transactions. Lastly, the label-printing utility, XLABEL.PRO, is explained.

Sorting for Common Interests

The basic facilities provided by the screen handler make it easy to construct programs with impressive user interfaces. The program in XCLUB.PRO maintains a database containing people's names, addresses, and interests, and searches the database for those people having a particular set of interests. The sample database is called XCLUB.DBA. The menu shown in Figure 3.8 is displayed when the program is executed.

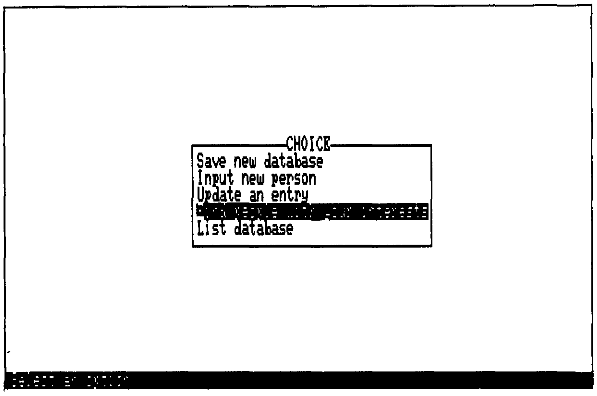
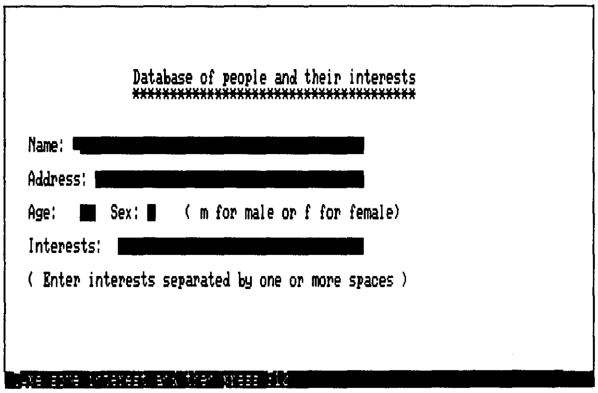


Figure 3.8: XCLUB.PRO's Menu

The first option, Save New Database, saves the sorted database in a disk file. When that option is selected, you are prompted for a file name, which is given the extension .DBA by default.

The last option, List Database, displays the current contents of the database on the screen.

If any of the remaining three options is selected, the screen shown in Figure 3.9 is displayed, with the bottom message line containing the appropriate Help information. This screen layout is specified in the screen-definition file XCLUB.SCR, ready for use by the screen handler.



```
Database of people and their interests
**************************************

Name:
Address:
Age:        Sex:     ( m for male or f for female)
Interests:
( Enter interests separated by one or more spaces )

[illegible]
```

Figure 3.9: XCLUB.SCR's Screen Layout

The Input New Person option allows the user to fill in the entries shown in Figure 3.9. Pressing F10 adds that person to the database in memory.

To update an Entry, you must fill in the *Name* field. The program then searches for the first entry in the database with that *Name* field value. If the entry can't be matched, the main menu is displayed. If a matching entry is found, the remaining fields in Figure 3.9 are filled in automatically; you can then modify them. When you're done, press F10 to return to the main menu.

The Find People With These Interests option allows you to type in a list of interests, separated by one or more spaces (for example, `golf fishing aerobics`). Press F10 to display successive screens of all the people in the database with those interests.

Using Screen Handlers in a Sample Program

This program uses many of the tools introduced so far, particularly the screen-handling facilities described in the section "Basic Use of the Screen Handler."

HNDBASIS.PRO is used with the addition of the database predicate *person(Name,Address,Age,Sex,Interests)* and the declaration of defining clauses for the remaining predicates used. The default clauses for *field_action, field_value,* and *noinput* remain the only defining clauses for those predicates.

XCLUB.PRO

```
check_determ code=3000

include "tdoms.pro"

domains
   FNAME = SYMBOL
   FNAMELIST = FNAME*
   TYPE = int(); str(); real()

domains
/* Domains for the demo */
   Name, Address = STRING
   Age = INTEGER
   Sex = m or f
   Interest = symbol
   Interests = Interest*
   FILE = textfile

database
/* Database declarations used in SCRHND */
   insmode                                        /* Global insert mode */
   actfield(FNAME)                                     /* Actual field */
   screen(SYMBOL,DBASEDOM)                  /* Saving different screens */
   value(FNAME,STRING)                             /* Value of a field */
   field(FNAME,TYPE,ROW,COL,LEN)                  /* Screen definition */
   txtfield(ROW,COL,LEN,STRING)
   windowsize(ROW,COL).
   notopline

/* Database predicates used in VSCRHND */
   windowstart(ROW,COL)
   mycursord(ROW,COL)

/* Database declarations used in LINEINP */
   lineinpstate(STRING,COL)
   lineinpflag

/* Local database */
   person(Name,Address,Age,Sex,Interests)

include "tpreds.pro"
include "menu.pro"
include "status.pro"
include "lineinp.pro"
include "filename.pro"
include "scrhnd.pro"
```

```
predicates
/* Predicates for the people demo */
   gsex(STRING,Sex)
   ginterests(STRING,Interests)
   gperson(Dbasedom)
   wperson(Dbasedom)
   listdba
   wr(DBASEDOM)
   process(INTEGER)
   nondeterm member(INTEREST,INTERESTS)

goal
   makewindow(77,7,0,"",0,0,24,80),
   makestatus(112,""),
   consult("xclub.scr"),
   consult("xclub.dba"),
   repeat,
   changestatus(" Select an option."),
   menu(10,25,7,7,
        ["Save new database",
         "Input new person",
         "Update an entry",
         "Find people with your interests",
         "List database"],
        "CHOICE",
        4,Ch),
   process(CH),CH = 0,!.

clauses
   member(X,[X|_]).
   member(X,[_|L]) :- member(X,L).

   field_action(_) :- fail.
   field_value(FNAME,VALUE) :- value(FNAME,VALUE),!.
   noinput(_) :- fail.

   process(0).
   process(1) :-
         changestatus("Type in a name for the database."),
         readfilename(10,10,7,7,dba,"xclub.dba",FILE),
         openwrite(textfile,FILE),
         writedevice(textfile),
         listdba,
         closefile(textfile).
   process(2) :-
         retract(value(_,_)),fail.
   process(2) :-
         createwindow(off),
         changestatus("Enter data on new person. Move cursor with arrows.
        F10:End"),
         scrhnd(off,KEY),not(KEY=esc),
         gperson(P),assertz(P),fail.
   process(2) :-
         removewindow.
   process(3) :-
         retract(value(_,_)),fail.
   process(3) :-
         createwindow(off),
         changestatus("To find an old record, give a name and press F10."),
        scrhnd(off,KEY1),not(KEY1=esc),
        value(f1,N),
        Name = N,
        person(Name,Ad,Al,K,I),
        wperson(person(Name,Ad,Al,K,I)),
        changestatus("Now you can modify the data. Press F10 to finish."),
        scrhnd(off,KEY2),not(KEY2=esc),
        retract(person(Name,Ad,Al,K,I)),
        gperson(P),
        asserta(P),
        removewindow,!.
   process(3) :-
        removewindow.
   process(4) :-
        retract(value(_,_)),fail.
   process(4) :-
        createwindow(off),
        changestatus("Enter interests and press F10"),
        scrhnd(off,KEY1),not(KEY1=esc),
        value(f5,S5), ginterests(S5,Interests),
        person(Name,Ad,Al,K,I),
        member(X,Interests),member(X,I),
        wperson(person(Name,Ad,Al,K,I)),
        changestatus("To inspect each matching entry, press F10."),
        scrhnd(off,KEY2),not(KEY2=esc),
        fail.
   process(4) :-
        removewindow.
   process(5) :-
        clearwindow,listdba.

/*************************************************************************
        Write and get data to and from the "value" predicate
*************************************************************************/

   wperson(_) :- retract(value(_,_)),fail.
   wperson(person(Name,Address,Age,Sex,Interests)) :-
        Name = S1,assertz(value(f1,S1)),
        Address = S2,assertz(value(f2,S2)),
        str_int(S3,Age),assertz(value(f3,S3)),
        gsex(S4,Sex),assertz(value(f4,S4)),
        ginterests(S5,Interests),assertz(value(f5,S5)).

   gperson(person(Name,Address,Age,Sex,Interests)) :-
        value(f1,S1), Name = S1,
        value(f2,S2), Address = S2,
        value(f3,S3), str_int(S3,Age),
        value(f4,S4), gsex(S4,Sex),
        value(f5,S5), ginterests(S5,Interests),!.

/*************************************************************************
   Conversions between a string and the corresponding domain
*************************************************************************/

   gsex("m",m).
   gsex("f",f).

   ginterests("",[]) :- !.
   ginterests(S,L) :- bound(S),fronttoken(S,",",S1),!,ginterests(S1,L).
   ginterests(S,[H|T]) :- bound(S),!,fronttoken(S,H,S1),ginterests(S1,T).
   ginterests(S,[H]) :- bound(H),!,H = S.
   ginterests(S,[H|T]) :- bound(H),
         ginterests(SS,T),concat(H,",",SSS),
         concat(SSS,SS,S).

/**********************************************************************
                          List the database
**********************************************************************/

   wr(X) :- write(X),nl.

   listdba :-
         person(A,B,C,D,E),
         wr(person(A,B,C,D,E)),
         fail.
   listdba.
```

Recording Sales Transactions

This section describes the program XSHOP.PRO, which can form the basis of an order-logging system for use in department stores and the like. When the program is executed, the predicate *scrhnd* in XSHOP.SCR creates the following screen.

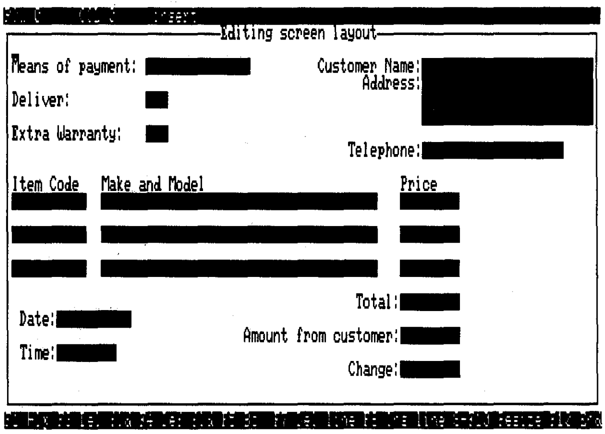


Figure 3.10: Sales Transaction Record

The program assumes that a customer can make three purchases at most; pay by cash, check, or credit card; have the items delivered; and take out an extra warranty agreement.

If Return is pressed with the cursor in the *payment* field, then a menu pops up showing the available range of payment methods. The user selects one of these from the menu, and the choice made is displayed in the *payment* field. Pressing Return in the *delivery* and *warranty* fields toggles between the words **yes** and **no**. The *date* and *time* fields display the date and time obtained from the PC's internal clock, with the *time* field continually updated. These two fields are no-input fields.

The top right-hand corner of the display contains the fields for the customer's name, address, and telephone number, which must be typed in by the user. (A complete sales system could use this information to generate supplemental forms, such as delivery notes and warranty records.)

To avoid the entry of incorrect item codes and prices, the *Make and Model* and *Price* fields are filled in automatically once the *Item Code* field has been filled in. If the correct item code is entered, the two corresponding fields are filled in automatically. Otherwise, a menu appears containing a list of the items available. Once the user selects an item, all three fields for that item are filled in by the program. Hence, all *Make and Model* and *Price* fields are no-input fields.

Of the remaining fields, the *Total* and *Change* fields are also no-input fields. *Total* is automatically updated by the program to reflect the total of the prices of all items entered so far. Once the user has entered the amount paid by the customer in the *Money* field, *Change* automatically shows the difference between the two amounts.

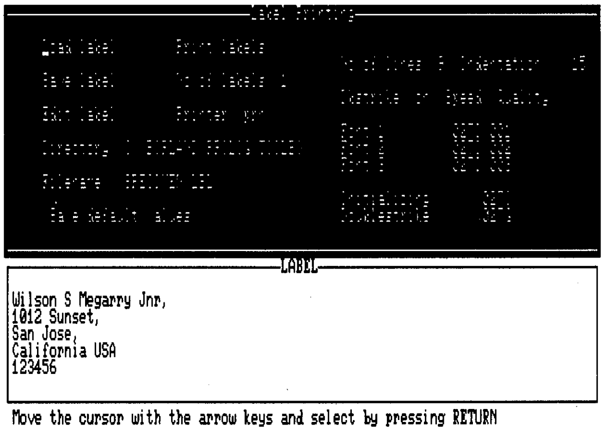
XSHOP.PRO

```
check_determ code=3000

include "tdoms.pro"

domains
   FNAME = SYMBOL
   TYPE = int(); str(); real()
   FILE = myprinter

database
/* Database declarations used in SCRHND */
   insmode                                            /* Global insertmode */
   actfield(FNAME)                                    /* Actual field */
   screen(SYMBOL,DBASEDOM)                            /* Saving different screens */
   value(FNAME,STRING)                                /* Value of a field */
   field(FNAME,TYPE,ROW,COL,LEN)                      /* Screen definition */
   txtfield(ROW,COL,LEN,STRING)
   windowsize(ROW,COL).
   notopline

/* Database declarations used in VSCRHND */
   windstart(ROW,COL)
   mycursord(ROW,COL)

/* Database declarations used in LINEINP */
   lineinpstate(STRING,COL)
   lineinpflag

/* SPECIFIC FOR THIS APPLICATION */
   payment(STRING)
   delivery
   warranty

clauses
   payment(cash).

include "tpreds.pro"
include "menu.pro"
include "lineinp.pro"
include "status.pro"
include "scrhnd.pro"

goal
   consult("xshop.scr"),
   makewindow(1,66,66,"Sales Transaction Record",0,0,24,80),
   makestatus(112,"Fill in the sales record and press F10 when finished",
              0,0,25,80),
   scrhnd(off,_).

predicates
   index(INTEGER,STRINGLIST,STRING)
   concatlist(STRINGLIST,STRING)

clauses
   index(1,[H|_],H) :- !.
   index(N,[_|T],X) :- N>1,N1 = N-1,index(N1,T,X).

   concatlist([],"").
   concatlist([H|T],S) :-
         concatlist(T,S1),concat(H,S1,S).

predicates
   nondeterm product(STRING,STRING,REAL)

clauses
   product("1111","Washing Machine",200.35).
   product("2222","Dishwasher",239.67).
   product("3333","Fridge and Freezer",456.78).
   product("4444","Radio",456.78).
   product("5555","Television",456.78).

clauses

/************************************************************************
                        Main routines
************************************************************************/
```

```
   noinput(payment).   noinput(delivery).        noinput(warranty).
   noinput(date).      noinput(time).            noinput(total).
   noinput(change).    noinput(make1).           noinput(make2).
   noinput(make3).     noinput(price1).          noinput(price2).
   noinput(price3).

/***********************************************************************
                        field_action
***********************************************************************/

predicates
   make(FNAME)

clauses
   field_action(delivery) :- retract(delivery),!.
   field_action(delivery) :- assertz(delivery).
   field_action(warranty) :- retract(warranty),!.
   field_action(warranty) :- assertz(warranty).
   field_action(payment) :- retract(payment(_)),fail.
   field_action(payment) :-
         cursor(R,C),
         LIST = ["Cash","Check","Credit card"],
         menu(R,C,7,7,LIST,"Please select method of payment",1,PayNo),
         index(Payno,LIST,STR),
         assertz(payment(STR)).
   field_action(item1) :- !,make(item1).
   field_action(item2) :- !,make(item2).
   field_action(item3) :- !,make(item3).

   make(FNAME) :- retract(value(FNAME,_)),fail.
   make(FNAME) :-
         cursor(R,C),
         findall(X,product(X,_,_),CODELIST),
         findall(X,product(_,X,_),LIST),
         menu(R,C,7,7,LIST,"Please select the product",1,Prod),
         index(Prod,CODELIST,CODE),
         assertz(value(FNAME,CODE)).

/***********************************************************************
                             field_value
***********************************************************************/

predicates
   price(FNAME,REAL)

clauses
   price(FNAME,PRICE) :-
         value(FNAME,CODE),product(CODE,_,PRICE),!.
   price(_,0).

   field_value(time,TIME) :- !,
         time(H,M,S,_),
         str_int(HS,H),str_int(MS,M),str_int(SS,S),
         concatlist([HS,":",MS,":",SS],TIME).

   field_value(date,DATE) :- !,
         date(D,M,Y),
         str_int(DS,D),str_int(MS,M),str_int(YS,Y),
         concatlist([DS,":",MS,":",YS],DATE).
   field_value(total,TotalS) :- !,
         price(item1,P1),
         price(item2,P2),
         price(item3,P3),
         Total = P1+P2+P3,
         str_real(TotalS,Total).

   field_value(change,CS) :- !,
         value(money,MM),!,str_real(MM,M),
         price(item1,P1),

         price(item2,P2),
         price(item3,P3),
         Total = M-(P1+P2+P3),
         str_real(CS,Total).

   field_value(item1,CODE) :- !,value(item1,CODE),!.
   field_value(item2,CODE) :- !,value(item2,CODE),!.
   field_value(item3,CODE) :- !,value(item3,CODE),!.

   field_value(make1,DESC) :- !,value(item1,Code),product(Code,Desc,_),!.
   field_value(make2,DESC) :- !,value(item2,Code),product(Code,Desc,_),!.
   field_value(make3,DESC) :- !,value(item3,Code),product(Code,Desc,_),!.

   field_value(price1,PRICEs) :- !,
         value(item1,Code),product(Code,_,PRICE),!,str_real(PRICEs,PRICE).

   field_value(price2,PRICEs) :- !,
         value(item2,Code),product(Code,_,PRICE),!,str_real(PRICEs,PRICE).
   field_value(price3,PRICEs) :- !,
         value(item3,Code),product(Code,_,PRICE),!,str_real(PRICEs,PRICE).

   field_value(payment,P) :- payment(P),!.

   field_value(delivery,yes) :- delivery,!.
   field_value(delivery,no) :- !.

   field_value(warranty,yes) :- warranty,!.
   field_value(warranty,no) :- !.

/* Catch other values from the database */
   field_value(Fn,X) :- value(Fn,X),!.
```

A Label-Printing Program

The final example in this section is a label-printing utility contained in XLABEL.PRO. Its screen-definition file is XLABEL.SCR, which uses *scrhnd* to produce the following screen.



```
Label Printing
Load Label      Print Labels
                                  No of lines 5  Indentation  15
Save Label      No of Labels 1
                                  [illegible]
Edit Label      Printer prn
                                  [illegible]
Directory [illegible]
Filename [illegible]
                                  [illegible]
 Save default values

LABEL
Wilson S Megarry Jnr,
1012 Sunset,
San Jose,
California USA
123456

Move the cursor with the arrow keys and select by pressing RETURN
```

Figure 3.11: Label-Printing Utility Screen

Pressing Return in the Edit Label field calls the *editmsg* standard predicate, so that the displayed label can be edited with the full Turbo Prolog editor. Press F10 to terminate editing; then save the label in a disk file by pressing Return with the cursor in the Save Label field. Retrieve a previously saved label by pressing Return with the cursor in the Load Label field. The Filename Label field displays the file name used for the last load or save operation and is a no-input field. The Directory field allows a default directory to be specified.

The Printer field specifies whether the printer is attached to PRN, COM1, or COM2. Selecting that field produces a menu, from which one of these three must be chosen. Once the number of labels to be printed has been set, printing begins when the Print Labels field is selected.

However, before printing any labels, the fields on the right-hand side of the screen must be adjusted to the desired printer characteristics. Thus, you should do the following:

- DoubleStrike (*dbstrike*) must be turned ON or OFF.
- The number of lines to be printed per label and the desired level of indentation must be set.

- A choice must be made between fast, medium, and letter-quality speeds of printing and the fonts to be used at each speed (specified with printer escape sequences).
- The appropriate Initializing and Doublestrike escape sequences must be entered if they are different from the defaults.

(These settings work with an Epson or Epson-compatible printer.)

XLABEL.SCR

```
check_determ code=2500

include "tdoms.pro"

domains
   FNAME = SYMBOL
   TYPE = int(); str(); real()
   FILE = myprinter

database
/* Database declarations used in SCRHND */
   insmode                                              /* Global insertmode */
   actfield(FNAME)                                           /* Actual field */
   screen(SYMBOL,DBASEDOM)                        /* Saving different screens */
   value(FNAME,STRING)                                   /* Value of a field */
   field(FNAME,TYPE,ROW,COL,LEN)                        /* Screen definition */
   txtfield(ROW,COL,LEN,STRING)
   windowsize(ROW,COL).
   notopline

/* Database declarations used in VSCRHND */
   windowstart(ROW,COL)
   mycursord(ROW,COL)

/* Database declarations used in LINEINP */
   lineinpstate(STRING,COL)
   lineinpflag

   label(STRING)
   dbstrike
   font(INTEGER)
   printer(STRING)

include "status.pro"
include "tpreds.pro"
include "menu.pro"
include "lineinp.pro"
include "filename.pro"
include "scrhnd.pro"
```

```
/**********************************************************************
                      Internal predicates
**********************************************************************/

predicates
   displbl
   printlabels
   change(DBASEDOM)
   filename(STRING)
   rdfile(STRING,STRING)
   index(INTEGER,STRINGLIST,STRING)
   nondeterm fonttext(INTEGER,STRING)
   nondeterm printers(STRING)
   nondeterm member(STRING,STRINGLIST)
   nondeterm for(INTEGER,INTEGER,INTEGER)
   write_n(INTEGER,CHAR)
   str_lines(STRING,STRINGLIST)
   setprintercodes
   printlabel(STRINGLIST)
   printercode(STRING)

clauses
   change(value(X,_)) :- retract(value(X,_)),fail.
   change(label(_)) :- retract(label(_)),fail.
   change(font(_)) :- retract(font(_)),fail.
   change(printer(_)) :- retract(printer(_)),fail.
   change(label(LBL)) :- !,assertz(label(LBL)),displbl.
   change(X) :- assertz(X).

   displbl :-
         label(LBL),!,
         shiftwindow(5),
         window_str(LBL),
         shiftwindow(1).

   filename(FILENAME) :-
         value(file,FNAME),!,
         cursor(ROW,COL),R1 = ROW+2,
         readfilename(R1,COL,7,7,lbl,FNAME,FILENAME),
         change(value(file,FILENAME)).

   rdfile(FILENAME,LABEL) :-
         file_str(FILENAME,LABEL),!.
   rdfile(FILENAME,_) :-
         makewindow(1,7,7,"",5,20,4,45),
         write(">> File not found: ",FILENAME),
         readkey(_),removewindow,fail.

   fonttext(1,"Fast").  fonttext(2,"Medium").  fonttext(3,"Quality").

   printers(prn).  printers(com1).  printers(com2).

   printlabels :-
         label(LBL),str_lines(LBL,LIST),
         printer(PRINTER),
         value(number,NN),str_int(NN,NOOFLABELS),
         openwrite(myprinter,PRINTER),
         writedevice(myprinter),
         setprintercodes,
	for(I,0,NOOFLABELS),
	printlabel(LIST),
	I> = NOOFLABELS-1,!,
	closefile(myprinter).

printlabel(LIST) :-
	value(indent,NN),str_int(NN,N),
	member(LINE,LIST),
	write_n(N,' '),write(LINE),nl,
	fail.
printlabel(LIST) :-
	listlen(LIST,LEN),
	value(labellines,TT),str_int(TT,TOT),!,
	SKIP = TOT-LEN,
	write_n(SKIP,'\n').

setprintercodes :-
	value(initcode,INIT),
	printercode(INIT),fail.
setprintercodes :-
	dbstrike,
	value(dbstrikecode,DBSTRIKE),
	printercode(DBSTRIKE),fail.
setprintercodes :-
	font(N),str_int(NO,N),
	concat("font",NO,FRONT),
	value(FRONT,FRONTCODE),
	printercode(FRONTCODE),fail.
setprintercodes.

printercode("") :- !.
printercode(CODE) :-
	frontchar(CODE,'\\',REST),
	fronttoken(REST,NUM,RESTCODE),
	str_int(NUM,CHI),
	char_int(CH,CHI),
	write(CH),!,
	printercode(RESTCODE).
printercode(CODE) :-
	frontchar(CODE,CH,REST),
	write(CH),
	printercode(REST).

index(1,[H|_],H) :- !.
index(N,[_|T],X) :- N>1,N1 = N-1,index(N1,T,X).

member(X,[X|_]).
member(X,[_|L]) :- member(X,L).

for(I,I,_).
for(I,A,B) :- B>A,A1 = A+1,for(I,A1,B).

write_n(0,_) :- !.
write_n(N,CH) :- N>0,write(CH),N1 = N-1,write_n(N1,CH).

str_lines("",[]) :- !.
str_lines(STR,[H|T]) :-
	search_char('\n',STR,0,N),
	frontstr(N,STR,H,R),
      frontchar(R,_,R1),!,
      str_lines(R1,T).
  str_lines(STR,[STR]).

/*********************************************************************
               Screen-handling predicates
*********************************************************************/

  noinput(load).      noinput(save).           noinput(saveconfig).
  noinput(print).     noinput(printer).        noinput(font).
  noinput(edit).      noinput(dir).            noinput(dbstrike).

  field_action(load) :-
      cursor(ROW,COL),R1 = ROW+2,
      readfilename(R1,COL,7,7,lbl,"",FILENAME),
      rdfile(FILENAME,LABEL),
      change(value(file,FILENAME)),
      change(label(LABEL)).
  field_action(save) :-
      filename(FILENAME),
      label(LABEL),!,
      file_str(FILENAME,LABEL).
  field_action(edit) :-
      label(LABEL),
      shiftwindow(5),
      editmsg(LABEL,LABEL1,"edit","","",0,"",RET),
      shiftwindow(1),
      refreshstatus,
      RET><1,!,
      change(label(LABEL1)).
  field_action(edit) :- displbl.
  field_action(dir) :- cursor(ROW,COL),setdir(ROW,COL).
  field_action(file) :- filename(_).
  field_action(print) :- printlabels.
  field_action(dbstrike) :- retract(dbstrike),!.
  field_action(dbstrike) :- assertz(dbstrike).
  field_action(font) :-
      cursor(ROW,COL),
      findall(X,fonttext(_,X),LIST),
      menu(ROW,COL,7,7,LIST,"Choose font",0,FRONT),
      FRONT><0,
      change(font(FRONT)).
  field_action(printer) :-
      cursor(ROW,COL),
      findall(X,printers(X),LIST),
      menu(ROW,COL,7,7,LIST,"Choose printer",0,NR),
      index(NR,LIST,PRINTER),
      change(printer(PRINTER)).
  field_action(saveconfig) :- save("xlabel.dba").

  field_value(dir,DISK) :- !,disk(DISK).
  field_value(dbstrike,on) :- dbstrike,!.
  field_value(dbstrike,off) :- !.
  field_value(font,FRONT) :- !,font(NR),fonttext(NR,FRONT),!.
  field_value(printer,X) :- !,printer(X),!.
  field_value(FNAME,VAL) :- value(FNAME,VAL),!.

goal
  makewindow(5,7,7,"LABEL",15,0,9,80),
makewindow(1,7,7,"Label Printing",0,0,15,80),
makestatus(112," Move the cursor with the arrow keys and select
        by pressing RETURN"),
consult("xlabel.dba"),
displbl,
scrhnd(off,_).
```

More Advanced Uses of the Screen Handler

Specifying New Special Keys

Within SCRHND.PRO and VSCRHND.PRO (the files containing the definition of *scrhnd*), the action taken when a key is pressed is controlled by the tool predicate *scr*. Thus, there are a number of clauses of the form

```
scr(char(C)) :- . . . . . . . . . .

scr(up) :- . . . . . . .

scr(fkey(10)) :- . . . . . .

scr(esc) :- . . . . . .
```

Defining a new special key simply involves adding values to these clauses. For example, the + key could be given a special meaning as follows:

```
scr(char('+')) :- value(f1, Val1), value(f2, Val2),
     str_int(Val1, NumVal1),
     str_int(Val2, NumVal2),
     NumSum = NumVal1 + NumVal2,
     str_int(Sum, NumSum),
     assertz(value(total, Sum)).
```

Associating Help Text With a Field

If the cursor is currently in a data field, the predicate *actfield* binds its single parameter to the name of that field. Field actions are then initiated by a clause of the form:

```
scr(cr) :- actfield(FieldName),                    /* Name of active field */
     field_action(FieldName),                      /* Action for that field */
     fail.

scr(cr) :- scr(tab).                               /* Otherwise go to the next field */
```

You can use this mechanism to associate Help text with a field. Each field that will be associated with a Help facility should have a corresponding file containing the appropriate text. For the field *time,* you could have a file called TIME.HLP that contains the text

```
Please specify the time
in the form hh:mm:ss,
taking care to use eight
characters total in all and to
separate the hours, minutes,
and seconds values with colons.
```

Now you can associate the field with the Help file by adding the following clause for *scr* to a copy of SCRHND.PRO:

```
   scr(fkey(1)) :- actfield(FieldName),
        concat(FieldName,".HLP",HelpFileName),
        file_str(HelpFileName,OurHelpText),
        makewindow(. . . . . .),
        display(OurHelpText),
        removewindow.
```

When the cursor is in a field that has a .HLP file defined, pressing F1 results in the contents of that file being *display*ed in the given window.

Alternatively, *displayhelp* can be used to make the Help messages context sensitive (see "Context-Sensitive Help" in Chapter 2) by defining the form

```
   scr(fkey(1)) :-
        actfield(FieldName),displayhelp(FieldName).
```

where *displayhelp* is an internal predicate defined in HELPDEF.PRO that looks in the appropriate database for that *FieldName* and displays the text for that Help context.

Adding Your Own Screen-Definition Types

The program file SCRHND.PRO permits screen definitions to contain fields of three different types: INTEGER, REAL, and STRING. These are controlled by the predicates *types* and *valid* and the domain TYPE. The declarations and clauses that contain these domains and predicates are as follows:

```
domains
   TYPE = int(); str(); real()
```

```
predicates
   valid(FNAME,TYPE,STRING)
   types(INTEGER,TYPE,STRING)

clauses
   valid(_,str,_).
   valid(_,int,STR)  :- str_int(STR, _).
   valid(_,real,STR) :- str_real(STR, _).

   types(1,int,"integer").
   types(2,real,"real").
   types(3,str,"string").
```

As an exercise, let's add a new type—DATE—to the collection of field types known to the screen handler. It's done simply by augmenting the declarations and clauses. Entries of this new type should be a date in the format mm/dd/yy (05/23/87 would represent May 23, 1987, for example).

The text string displayed at the top of the screen when the cursor is in a field (during execution of SCRDEF.PRO or if the first parameter of *scrhnd* is ON) is associated with that field by *types*. Thus, you need to add

```
   types(4,date,"date").
```

to the clauses for *types*.

Before the cursor is allowed to leave a given field, the value in that field is verified to be of the correct type by the predicate *valid*.

```
   valid(FieldName, FieldValueType, EntryString)
```

succeeds only if the *EntryString* that the user has inserted in *FieldName* has a value of the correct *FieldValueType*. (You can specify more precise validation—for example, file names entered could be checked in the directory to confirm their existence.)

To verify entries of date type, add a clause like

```
   valid(_, date, EntryString) :-
         fronttoken(EntryString, Month, Rest1),
         str_int(Month, _),
         fronttoken(Rest1,_,Rest2),
         fronttoken(Rest2, Day, Rest3),
         str_int(Day, _),
         fronttoken(Rest3, _, Year)
         str_int(Year, _).
```

Finally, augment the declaration of the TYPE domain to **include** objects of type DATE:

```
domains
   TYPE = int(); str(); real(); date()
```

These additions to SCRHND allow fields of DATE type. The same alterations must be made to SCRDEF.PRO, except for *valid*, which is not used in the latter program.

Using Several Screen Layouts Interchangeably

SCRHND.PRO contains the tool predicates *screen* and *shiftscreen*, which allow you to use several screen layouts interchangeably. They are declared as follows:

```
database
   screen(SYMBOL, DBASEDOM)

predicates
   shiftscreen(SYMBOL)
```

Suppose you want to use two different screen layouts, referred to by the SYMBOLic names *screen1* and *screen2*. The basic idea is to move from one screen to the other by a call to *shiftscreen* of the form

```
   shiftscreen(NewScreen)
```

If you have been using *screen1* and *NewScreen* is bound to *screen2*, *scrhnd* will operate with *screen2* after the call.

Screen-definition files contain facts for the database predicates *textfield*, *field*, and *windowsize*. These facts are normally consulted before a call to *scrhnd*. In order to use *shiftscreen*, store the facts for both screen definitions in a file of facts for *screen*. If *screen1* were defined by

```
   field("input1",str,9,27,17)
   txtfield(7,27,17,"Input on screen 1")
   windsize(20,77)
```

and *screen2* were defined by

```
   field("input2",int,9,27,25)
   txtfield(7,27,16,"Screen Two Input")
   windsize(20,77)
```

then we would store the following facts for *screen*:

```
   screen(screen1,field("input1",str,9,27,17)).
   screen(screen1,txtfield(7,27,17,"Input on screen 1")).
   screen(screen1,windsize(20,77)).
   screen(screen2,field("input2",int,9,27,25)).
   screen(screen2,txtfield(7,27,16,"Screen Two Input")).
   screen(screen2,windsize(20,77)).
```

The definition of *shiftscreen* shows how everything will work:

```
   shiftscreen(_) :- retract(field(_,_,_,_,_)),fail.
   shiftscreen(_) :- retract(txtfield(_,_,_,_)),fail.
   shiftscreen(_) :- retract(windsize(_,_)),fail.
   shiftscreen(NAME) :- screen(NAME,TERM),assertz(TERM),fail.
   shiftscreen(_).
```

As with all the Toolbox tools, you aren't limited by *screen* and *shiftscreen*. The straightforward use of *consult* is a perfectly viable alternative.

Creating New Screen Definitions from Old

The manner in which screen definitions are stored makes it possible to combine several screen definitions to form a new one. For example, the following Prolog fragment combines the definitions in GOODS.SCR, ORDERS.SCR, CUSTOMER.SCR, and INF.SCR into a single screen definition in INVOICE.SCR.

```
goal
   run.

clauses
   run :- retract(_),fail.
   run :- consult("goods.scr"),regscreen(goods),fail.
   run :- consult("orders.scr"),regscreen(orders),fail.
   run :- consult("customer.scr"),regscreen(customer),fail.
   run :- consult("inf.scr"),regscreen(inf),fail.
   run :- save("invoice.scr").

   regscreen(NAME) :-
         retract(field(A,B,C,D,E)),
         assertz(screen(NAME,field(A,B,C,D,E))),fail.
   regscreen(NAME) :-
         retract(txtfield(A,B,C,D)),
         assertz(screen(NAME,txtfield(A,B,C,D))),fail.
   regscreen(NAME) :-
         retract(windowsize(A,B,C,D)),
         assertz(screen(NAME,windowsize(A,B,C,D))),fail.
   regscreen(_).
```

Printing Formatted Reports

You can write a Turbo Prolog program that will enter information into a form, such as the one you get from the tax bureau every year, but it means experimenting with *write* statements. With the Toolbox, you can use

SCRDEF.PRO to define a screen layout like the tax form and then use the tool predicate *report* to actually print the current information on the forms.

The *report* predicate is contained in REPORT.PRO, which comments out the *write* standard predicate call in the goal. This avoids everything hanging if you attempt to run the program without the appropriate printer or printer driver card connected to your computer.

Having *consult*ed a screen-definition file, a call of the form

```
report(NoOfLinesRequired)
```

sends *NoOfLinesRequired* lines of the corresponding layout (including values entered by the user) to the current *outputdevice*.

XREPORT.PRO is a revamped version of XSHOP.PRO (see "Recording Sales Transactions"). It uses *report* to print a copy of the invoice created during execution of XSHOP.PRO. To make the printed output attractive, XSHOP.SCR also has been revamped; the new layout is contained in XREPORT.SCR.

XREPORT.PRO

```
include "tdoms.pro"

domains
   FNAME = SYMBOL
   TYPE = int(); str(); real()
   FIELD = field(STRING,COL)
   FIELDLIST = FIELD*

database
/* Database declarations used in SCRHND */
   insmode                                          /* Global insertmode */
   actfield(FNAME)                                  /* Actual field */
   screen(SYMBOL,DBASEDOM)                          /* Saving different screens */
   value(FNAME,STRING)                              /* Value of a field */
   field(FNAME,TYPE,ROW,COL,LEN)                    /* Screen definition */
   txtfield(ROW,COL,LEN,STRING)
   windowsize(ROW,COL).
   notopline

/* Database declarations used in SCRHND */
   windstart(ROW,COL)
   mycursord(ROW,COL)

/* Database declarations used in LINEINP */
   lineinpstate(STRING,COL)

/* SPECIFIC FOR THIS APPLICATION */
   payment(STRING)
   delivery
   warranty

include "tpreds.pro"
include "report.pro"

goal
   consult("xreport.scr"),
   makewindow(1,31,0,"",0,0,25,80),
   write("\t\tRemove the comment in the goal to send to the printer\n\n"),
/* writedevice(printer), */
   report(20).

predicates
   nondeterm product(STRING,STRING,REAL)

clauses
   product("1111","Washing Machine",200.35).
   product("2222","Dishwasher",239.67).
   product("3333","Fridge and Freezer",456.78).
   product("4444","Radio",456.78).
   product("5555","Television",456.78).
```

predicates
```
   index(INTEGER,STRINGLIST,STRING)
   concatlist(STRINGLIST,STRING)
```

clauses
```
   index(1,[H|_],H) :- !.
   index(N,[_|T],X) :- N>1,N1 = N-1,index(N1,T,X).

   concatlist([],"").
   concatlist([H|T],S) :-
        concatlist(T,S1),concat(H,S1,S).
```

clauses

```
/***********************************************************************
                              field_value
***********************************************************************/
```

predicates
```
   price(FNAME,REAL)
```

clauses
```
   price(FNAME,PRICE) :-
        value(FNAME,CODE),product(CODE,_,PRICE),!.
   price(_,0).

   field_value(time,TIME) :- !,
        time(H,M,S,_),
        str_int(HS,H),str_int(MS,M),str_int(SS,S),
        concatlist([HS,":",MS,":",SS],TIME).

   field_value(date,DATE) :- !,
        date(D,M,Y),
        str_int(DS,D),str_int(MS,M),str_int(YS,Y),
        concatlist([DS,":",MS,":",YS],DATE).

   field_value(total,TotalS) :- !,
        price(item1,P1),
        price(item2,P2),
        price(item3,P3),
        Total = P1+P2+P3,
        str_real(TotalS,Total).

   field_value(change,CS) :- !,
        value(money,MM),!,str_real(MM,M),
        price(item1,P1),
        price(item2,P2),
        price(item3,P3),
        Total = M-(P1+P2+P3),
        str_real(CS,Total).

   field_value(item1,CODE) :- !,value(item1,CODE),!.
   field_value(item2,CODE) :- !,value(item2,CODE),!.
   field_value(item3,CODE) :- !,value(item3,CODE),!.

   field_value(make1,DESC) :- !,value(item1,Code),product(Code,Desc,_),!.
   field_value(make2,DESC) :- !,value(item2,Code),product(Code,Desc,_),!.
   field_value(make3,DESC) :- !,value(item3,Code),product(Code,Desc,_),!.
   field_value(price1,PRICEs) :- !,
        value(item1,Code),product(Code,_,PRICE),!,str_real(PRICEs,PRICE).
   field_value(price2,PRICEs) :- !,
        value(item2,Code),product(Code,_,PRICE),!,str_real(PRICEs,PRICE).
   field_value(price3,PRICEs) :- !,
        value(item3,Code),product(Code,_,PRICE),!,str_real(PRICEs,PRICE).

   field_value(payment,P) :- payment(P),!.

   field_value(delivery,yes) :- delivery,!.
   field_value(delivery,no) :- !.

   field_value(warranty,yes) :- warranty,!.
   field_value(warranty,no) :- !.

/* Catch other values from the database */
   field_value(Fn,X) :- value(Fn,X),!.

   value("name","J B Gruntfuttock").
   value("street","3 Railway Terrace").
   value("city","Seattle").
   value("state","Washington").
   value("tel","222 333 4444").
   value("item1","1111").
   value("item2","2222").
   value("item3","3333").
   value("money","1000.00").
   payment("Check").
   delivery.
```

C H A P T E R

4

Graphics Tools

This chapter describes tools that help you construct programs to graphically represent information, such as financial statements and market research results. These tool predicates also allow you to show numerical information as bar charts, pie charts, and graphs.

The graphics tools described here use the domains and predicates defined in the files GDOMS.PRO and GPREDS.PRO (and, in certain circumstances, in GGLOBS.PRO). They must therefore be **included** in any program that uses them. (If you're unfamiliar with modular programming, refer to Appendix A, "Compiling a Project." It discusses project [.PRJ] and object [.OBJ] files.)

NOTE: These programs work on PCs with the Color Graphics Adapter (CGA) or Enhanced Graphics Adapter (EGA). They do not work on Hercules monochrome graphics cards.

Coping with Different Coordinate Systems

This section covers virtual-screen coordinates, low-level tool predicates (using virtual coordinates), and scale definition.

Virtual-Screen Coordinates

Turbo Prolog's standard predicates for graphics use virtual-screen coordinates. When the entire monitor screen is used, the virtual-screen coordinate system refers to the upper left-hand corner as (0,0) and the lower right-hand corner as (31999,31999). These coordinates are automatically scaled down to the actual pixel (picture element) positions on the monitor, whether the graphics card driving it provides 320 X 200, 640 X 200, or 640 X 350 pixels.

All the following standard predicates in Turbo Prolog take virtual-screen coordinates as parameters:

```
line(Row1,Column1,Row2,Column2,Color)
dot(Row,Column,Color)
penpos(Row,Column,Direction)
forward(Step)
back(Step)
```

If, for example, the Turbo Prolog standard predicate *dot* is called, as in

```
dot(16000,16000,1)
```

which uses the entire screen for graphics display, a colored dot appears at the center of the screen.

When the graphics standard predicates are used in a window, virtual coordinates within that window are relative to the upper left-hand corner of the window. This corner is regarded as coordinate position (0,0). If, for example, an unframed window is created with 12 rows and 80 columns, the upper left-hand corner of the window is at (0,0) and the lower right-hand corner is at [(32000X12/25)-1,31999]; that is, (15359,31999).

Low-level Tool Predicates and Virtual Coordinates

There are a number of low-level graphics tool predicates that are used to implement the higher level ones. These include tool predicates that plot points, shade shapes, and draw boxes, ellipses, and sectors. This section explains four of the most useful predicates, which are implemented as external predicates coded in C. Therefore, programs that use them need to be compiled and linked as projects. (See Appendix A.) Since GGLOBS.PRO contains all the relevant global declarations, it lists *all* the tool predicates supplied in GRAPHICS.OBJ, not just those presented here.

The Predicate plot

The call to the predicate plot should not contain any flow pattern or language declaration

The first tool, a more general plotting tool predicate than *dot,* is called *plot.* It is declared as follows:

```
plot(VROW,VCOL,COLOR,SIZE,KIND) - (i,i,i,i,i) language c
```

A call of the form

```
plot(Vrow,Vcol,Color,Size,Shape) - (i,i,i,i,i) language c
```

plots a pixel, a dot inside a box frame, a filled-in box, or the letter X at the virtual point (*Vrow,Vcol*). It plots the element in the named *Color,* with the given *Shape* according to the value to which *Shape* is bound:

0 One pixel is drawn.
1 A dot is drawn inside a box.
2 A box is drawn and filled.
3 The letter X is drawn.

The range for *Vrow, Vcol,* and *Vsize* is 0 to 31999 in each case.

The Predicate box

The tool predicate *box* draws a rectangle on the screen and is declared as follows:

```
box(VROW,VCOL,VROW,VCOL,COLOR,COLOR,FILL) - (i,i,i,i,i,i,i) language c
```

A call of the form

```
box(Row1,Col1,Row2,Col2,LineColor,FillColor,Fill)
```

with *Row1, Row2, Col1,* and *Col2* in the range 0 to 31999, draws a box according to the value to which *Fill* is bound:

0 A box is drawn with color *LineColor* but not filled.
1 A box is drawn with color *LineColor* and filled with the color *FillColor.*

The Predicate ellipse

ellipse is declared in a similar way:

```
ellipse(VROW,VCOL,VRADIUS,REAL,COLOR,COLOR,FILL) - (i,i,i,i,i,i,i) language c
```

In this case, the call

```
ellipse(Row,Column,VerticalRadius,Ratio,FrameColor,Color,Fill)
```

draws an ellipse, provided that the following are true: *Row, Column,* and *VerticalRadius* are in the range 0 to 31999; *Ratio* is a real number between 0 and 1 representing the ratio of the horizontal radius of the ellipse to its vertical radius. If *Fill* is bound to 0, an ellipse is drawn but not filled in; if *Fill* is 1, then an ellipse is drawn and filled.

The Predicate sector

Next in this group of low-level tool predicates comes *sector,* which draws the outline of a sector of a circle. It is declared as follows:

```
sector(VROW,VCOL,VRADIUS,INCREMENT,DEGREES,DEGREES,COLOR,COLOR,FILL)
      - (i,i,i,i,i,i,i,i) language c
```

A call takes the form

```
sector(Row,Col,Radius,Increment,StartAng,EndAng,BorderColor,Fill,ColorFill)
```

Row and *Column* should be bound to values in the range 0 to 31999 and *Increment* bound to an angle in the range 0 to 360 degrees. *sector* draws that sector of a circle centered on (*Row,Col*) with radius *Radius* occupying the angle between *StartAng* and *EndAng*—specified in degrees. A horizontal radius from the center of the circle pointing toward the right-hand side of the display screen denotes zero degrees. The sector is drawn in the given *Color* and is filled if *Fill* is 1 and not filled if *Fill* is 0. If it is filled, then *Increment* specifies the size of the angle used to fill the sector. The specified triangle is drawn in these increments.

The Predicate lineShade

Finally, there is *lineShade,* declared as follows:

```
lineShade(VROW,VCOL,VROW,VCOL,VCOL,COLOR,KIND) - (i,i,i,i,i,i,i) language c
```

A call of the form

```
lineShade(Row1,Col1,Row2,Col2,EndLine,FillColor,Direction)
```

draws a line between the points (*Row1,Col1*) and (*Row2,Col2*), and shades an area on one side of the line. The shading is toward the top of the screen, toward the bottom, or toward the left or right side of the screen depending on the value of the *Direction* parameter:

0 Shade up
1 Shade down
2 Shade left
3 Shade right

The shading covers the area from the beginning line to the line indicated by the *EndLine* parameter. When shading is toward the left or right of the screen, *Endline* is the column number—from the Toolbox domain VCOL—specifying a vertical line at which shading stops.

When shading up or down, *Endline* is represented by a VROW value. Coercion between the domains VROW and VCOL is necessary (look at *lineShade*'s declaration). This is done by a code fragment of the form

```
. . .CoerceVariable = VROWvalue, lineshade(. . . . . . . .,CoerceVariable,. .),
```

Row1, Row2, Col1, Col2, and *EndLine* must all be in the range 0 to 31999. The coding for *Direction* is as follows:

0 Shade up to the row specified by *EndLine*
1 Shade down to the row specified by *EndLine*
2 Shade left to the column specified by *EndLine*
3 Shade right to the column specified by *EndLine*

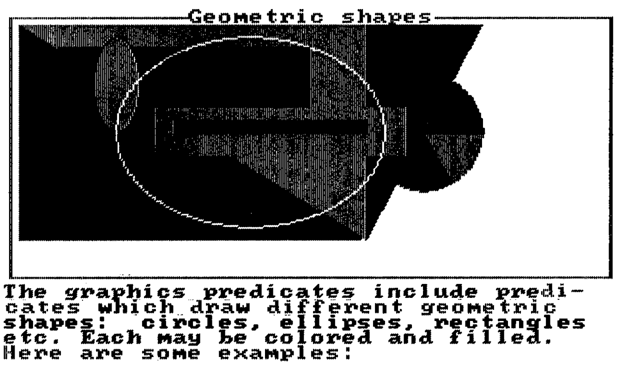
In all cases, coerce the value of *Endline* to a value in the VCOL domain.

Using Low-level Tool Graphics Predicates in a Sample Program

The following program is called XGEOMETR.PRO. It must be compiled as a project. The figure following the listing shows the screen display generated by the program. Since the display is set up for a color monitor, some of the color differentiation is lost on the black-and-white screen shot (Figure 4.1).

XGEOMTR.PRO

```
project "xgeometr"

include "tdoms.pro"
include "gdoms.pro"

database
   newsline(string)

include "gglobs.pro"
include "tpreds.pro"
include "gpreds.pro"

predicates
   message(string,string,string,string,string)
   wfs(char)
   wait(integer)
goal
   graphics(1,1,1),
   makewindow(1,6,6,"Geometric shapes",0,0,18,40),
   makewindow(3,7,0,"",18,0,6,40),
   makewindow(4,8,0,"SPACE BAR MESSAGE",24,5,1,20),

   message("This demo includes predicates which may",
          "be used to draw real art.","","",""),
   shiftwindow(1),
   LineShade(0,0,18000,18000,18000,1,1),
   LineShade(0,0,18000,18000,18000,3,0),
   LineShade(0,0,18000,18000,18000,0,2),
   LineShade(0,0,18000,18000,18000,2,3),
   LineShade(18000,18000,0,24000,0,1,0),
   wfs(_),
   message("The graphics predicates include",
          "predicates that draw different geometric",
          "shapes:  circles, ellipses, rectangles,",
          "etc. Each may be colored and filled.",
          "Here are some examples:"),
   shiftwindow(1),
   box(2000,2000,10000,15000,2,3,1),
   box(7000,7000,11000,20000,1,2,1),
   box(8000,8000,9000,18000,1,1,1),
   ellipse(5000,5000,4000,0.5,1,2,1),
   ellipse(9000,12000,8000,1.4,4,4,0),
   sector(9000,21000,5000,2,240,300,1,1,1),
   sector(9000,21000,5000,2,300,360,2,2,1),
   sector(9000,21000,5000,2,0,60,3,3,1),
   wfs(_).

clauses
   newsline("Press the space bar to continue.   ").
   message(S1,S2,S3,S4,S5) :-
         shiftwindow(Old),shiftwindow(3),
         clearwindow,
         attribute(1),write(S1,"\n"),
         attribute(2),write(S2,"\n"),
         attribute(3),write(S3,"\n"),
         attribute(1),write(S4,"\n"),
         attribute(2),write(S5,"\n"),
         shiftwindow(Old).

   wfs(C) :-
         shiftwindow(4),
         attribute(2),
         newsline(S),field_str(0,0,20,S),
         keypressed,readchar(C),!,
         field_str(0,0,20," "),shiftwindow(3).
   wfs(C) :- wait(2000),
         retract(newsline(String)),!,
         frontstr(1,String,F,Rest),
         concat(Rest,F,New),
         assertz(newsline(New)),
         wfs(C).

   wait(0) :- !. wait(N) :- N1 = N-1,wait(N1).
```



The graphics predicates include predi-
cates which draw different geometric
shapes: circles, ellipses, rectangles
etc. Each may be colored and filled.
Here are some examples:

Figure 4.1: Geometric Shapes

Defining Scales

Several methods can be used to address a given point on the screen:

- Virtual-screen coordinates
- Scaled coordinates
- Text coordinates
- Actual-screen coordinates referring directly to the pixels used by the hardware to form the display image

This section uses predicates found in GGRAPH.PRO and GDOMS.PRO to accomplish this task. Virtual graphics coordinates have the advantage of being hardware independent; however, you need to scale them to fit a specific application. For instance, to graph monthly sales for one particular year, a horizontal scale of 0 to 12 is probably the way to go. But that means you'll have to convert between the values 1,2,3,...,12 and the horizontal virtual-coordinate values that represent them.

As an alternative to virtual coordinates, the Toolbox provides several predicates to manage *scaled* coordinates. In scaled coordinate systems, the origin is the lower left-hand corner of the active window (which the *graphics* standard predicate initializes to the entire screen).

A *scaled* coordinate system is defined by the lowest and highest values to be displayed on the X axis and Y axis. As several scales may be in use simultaneously, each scale is identified by a *ScaleNumber* in the same way that windows in Turbo Prolog have a window number. Thus, the scale-defining equivalent of *makewindow* is *defineScale,* which takes the form

```
   defineScale(ScaleNo,Xmin,Xmax,Ymin,Ymax)
```

This defines a scale identified by the INTEGER parameter *ScaleNo,* in which values on the X axis run from *Xmin* to *Xmax* and on the Y axis from *Ymin* to *Ymax.* (Again, these are all INTEGER parameters.) When a *scale* is defined by *defineScale,* the database declarations

```
database
   scale(SCALENO,X,X,Y,Y)
   activeScale(SCALENO)
   axes(INTEGER,INTEGER,INTEGER,XMARKER,YMARKER,Col,Row,Col,Row)
```

must be **included** in your program. The tool predicate *shiftScale* corresponds to *shiftwindow* in that the call

```
shiftScale(ScaleNo)
```

utilizes the value of the SCALENO parameter to enable you to switch between scales.

Once a scale has been defined, any subsequent image is automatically scaled to fit the actual window and rescaled if the window is resized. Moreover, once defined, a scale can be modified easily using the database predicates shown. For example, the following goal modifies the currently active scale so that the range on the X axis is expanded to twice the original range:

```
goal
   activeScale(N),
   retract(scale(N,Xmin,Xmax,Ymin,Ymax)),!,
   NewXmax = 2*Xmax,
   assertz(scale(N,Xmin,NewXmax,Ymin,Ymax)).
```

The tool predicate *scalecursor* is used to position the cursor in scaled graphics. Its declaration takes the form

```
scalecursor(X,Y)
```

The call

```
scalecursor(23,51)
```

places the cursor (that is, starts the text) at the scaled coordinate position (23,51).

scalePlot is a tool predicate contained in GGRAPH.PRO and plots points when scaled graphics are used. Its declaration takes the form

```
scalePlot(X,Y,COLOR)
```

so that the call

```
scalePlot(100,200,3)
```

plots a point at position (100,200) according to the scale currently active and in color 3.

Another tool predicate, *scaleLine,* plots lines in scaled graphics. Its declaration takes the form

```
scaleLine(X,Y,X,Y,COLOR)
```

In a call of the form

```
scaleLine(X1,Y1,X2,Y2,Color)
```

(X1,Y1) is the starting point of the line, *(X2,Y2)* is the end point of the line, and *Color* determines the color of the line.

A more complicated image in a scaled display can be represented as a list of points belonging to the domain DRAWING, which is defined in GDOMS.PRO as

```
domains
   DRAWING = POINT*
   POINT = p(X,Y)
   X,Y = REAL
```

So, for example, the following DRAWING represents a square:

```
[p(20,30),p(50,30),p(50,60),p(20,60)]
```

An actual image is drawn from an element of the DRAWING domain by the *scalePolygon* tool predicate. Its declaration takes the form

```
scalePolygon(COLOR,DRAWING)
```

The points specified by the DRAWING are plotted and connected to form a polygon in the specified COLOR. For example,

```
scalePolygon(3,[p(3,1),p(5,3),p(3,5),p(1,3)])
```

draws a diamond shape on the screen, scaled to fit the currently active scale.

The tool predicate *draw* is used to draw a scaled polygon belonging to the domain DRAWINGS. An object from the domain DRAWINGS is a list of colored polygons in which the functor *d* binds a color to a DRAWING, which is itself a polygon:

```
X,Y = REAL
POINT = p(X,Y)
DRAWING = POINT*
D = d(COLOR,DRAWING)
DRAWINGS = D*
```

For example,

```
draw([d(2,[p(70,100),p(90,195),p(100,230)]),
```

produces a triangle.

Drawing Axes on the Screen

The Toolbox provides four tool predicates that have to do with drawing axes on a graphics display screen: *makeAxes, refreshAxes, modifyAxes,* and *axislabels*. All rely on a database predicate definition of the form

```
axisPair(INTEGER,INTEGER,INTEGER,XMARKER,YMARKER,COL,ROW,COL,ROW)
```

In the entry

```
axisPair(AxesPairNo,AxesWindowNo,GraphWindowNo,
         Xmarkers,Ymarkers,Left,Bottom,Right,Top)
```

AxesPairNo is the number of the scaled coordinate system used when you need to refer to these axes again, and they are drawn in window number *AxesWindowNo*.

When axes are defined using *makeAxes,* the first step on execution of the *makeAxes* call is drawing the axes in the currently active window. A new window is then created inside that active window so that subsequent images in the coordinate system are automatically clipped if the image being drawn exceeds the ranges for the currently active scale. The number of this window is returned in *GraphWindowNo*. *Left, Right, Bottom,* and *Top* specify the number of character positions that are to be left between the edges of this window and the border of the screen.

The markings on the axes are defined by two markers of the form

```
marker(Unit,Formatspecifier,FieldWidth)
```

These markers define the following:

- what size the numbers on a numbered interval (*Unit*) on the relevant axis are
- whether the numbering is in decimal or exponential form (that is, *FormatSpecifier* equals **d** or **e**, respectively)
- how many character positions on the screen each such number may occupy

The markers on the axes are modified using

```
modifyAxes(AxisPairNo,marker(Xunit,Xform,XWidth),
          marker(Yunit,Yform,YWidth))
```

This modifies the relevant *axisPair* database entry.

A set of axes is refreshed by giving a goal of the form

```
refreshAxes(AxisPairNo)
```

The X and Y axes are labeled by calling

```
axisLabels(window#,XaxisText,YaxisText)
```

The form of the predicate *makeAxes* consists of

```
makeAxes(AxesNo,AxesWindowNo,GraphWindow,Xmarkers,
          Ymarkers,Left,Bottom,Right,Top)
```

Using Scales and Axes in a Sample Program

The following sample program, XGRAPH.PRO, creates two windows, each used for a different graph. The first window, window number 1, is used with a scale defined as

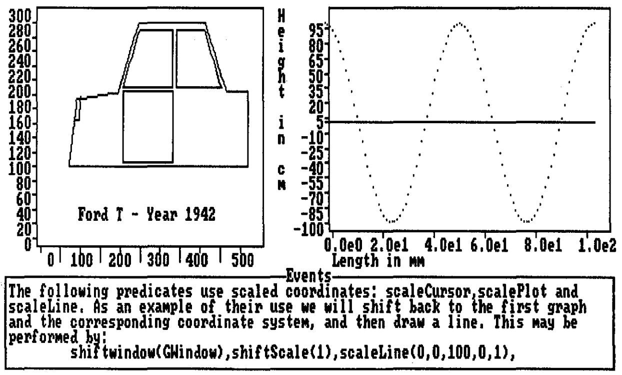
```
definescale(1,0,100,-100,100)
```

This scale is subsequently referred to as scale number 1. Then the axes are defined with *AxisPairNo* set to 1, and the corresponding internal window is also given the window number 1:

```
shiftwindow(1),
makeAxes(1,_,Gwindow,marker(10,d,2),marker(10,d,3),2,3,2,1)
```

(It is only for convenience that the corresponding window, scale, and axis pairs have the same reference number in this example.)

The demo also illustrates how to modify axes so they can reflect changes in the scales. This facility is used in the program to implement zooming capabilities and the like.



```
The following predicates use scaled coordinates: scaleCursor,scalePlot and
scaleLine. As an example of their use we will shift back to the first graph
and the corresponding coordinate system, and then draw a line. This may be
performed by:
        shiftwindow(GWindow),shiftScale(1),scaleLine(0,0,100,0,1),
```

Figure 4.2: Scales and Axes in a Screen

XGRAPH.PRO

```
code=3000
include "tdoms.pro"
include "gdoms.pro"
database
   Scale(ScaleNo,x,x,y,y)
   activeScale(ScaleNo)
   axes(integer,integer,integer,xmarker,ymarker,col,row,col,row)
   newsline(string)                                        /* Used by this demo only */

include "gglobs.pro"
include "tpreds.pro"
include "gpreds.pro"
include "ggraph.pro"

predicates
   element(string,drawing)
   process(char)
   zoom(x,x,y,y)
   message(string,string,string,string,string)
   wfs(char)
   wait(integer)
   function(x)
```

```
goal
   assertz(newsline("Press the space bar     ")),
   graphics(2,1,1),
   makewindow(1,11,1,"Window used for Graph 1",0,36,18,44),
   makewindow(2,12,1,"Window used for Graph 2",0,0,18,35),
   makewindow(3,13,1,"Events",18,0,7,80),
   makewindow(4,7,0,"SPACE BAR MESSAGE",24,30,1,20),

   message("The first graph is used for coordinates, which",
           "on the X axis run from 0 to 100 and on the Y axis",
           "run from -100 to +100.  This is accomplished by",
           "defining a 'scale'. In this way the graph is",
           "automatically scaled to fit the actual window.",
           "\tdefineScale(1,0,100,-100,100)"),
   defineScale(1,0,100,-100,100),
   wfs(_),
   message("Now we can draw axes in the actual window",
           "corresponding to that scale:",
           "",
           "\tmakeAxes(1,Ano,Gno,marker(10,d,2),marker(10,d,3),1,1,1,1)",
           "",
           ""),
   shiftwindow(1),
   removewindow,
   makewindow(1,11,0,"Graph 1",0,36,18,44),

   makeAxes(1,                                       /* Axes pair 1 */
            _,                                       /* Axes Window No. */
            GWindow,                                 /* Graphics Window No. */
            marker(10,d,2),
            marker(10,d,3),
            2,                                       /* Left */
            3,                                       /* Bottom */
            2,                                       /* Right */
            1),                                      /* Top */
   wfs(_),
   message("The axes may be labeled using, for example,",
           "",
           "\taxisLabels(1,\"Length in mm\",\"Height in cm\")",
           "",
           ""),
   axisLabels(1,"Length in mm","Height in cm"),
   wfs(_),
   message("Any output may be directed to the graphics window by:",
           "",
           "\tshiftwindow(GraphWindowNo)",
           "\t...draw function output....",
           ""),
   shiftwindow(GWindow),
   function(0),
   wfs(_),
   message("Normal text can be written in the graph area.",
           "",
           "\tshiftwindow(Gno),cursor(0,0),write(\"F(X) = 100*cos(6.28*X/50)\"),",
           "",
           ""),
   shiftwindow(GWindow),
   cursor(0,0),write("F(X) = 100*cos(6.28*X/50)"),
   wfs(_),
message("The markings of the axes can be modified. Note",
        "that the intervals have increased, and that",
        "exponential notation is used on the Y axis instead of",
        "decimal notation, due to the calls:",
        "",
        "\tmodifyAxes(1,marker(20,e,4),marker(15,d,4)), refreshAxes(1),"),
modifyAxes(1,marker(20,e,5),marker(15,d,4)),
refreshAxes(1),
axisLabels(1,"Length in mm","Height in cm"),
shiftwindow(GWindow),
function(0),
wfs(_),
message("Window 2 is to be used for another graph--a drawing--",
        "that is drawn relative to a new coordinate system.",
        "This requires two calls:",
        "",
        "\tdefineScale(2,0,500,0,300)",
        "\tmakeAxes(2,_,GWindow2,marker(50,d,3),marker(20,d,3),1,1,2,1),"),
shiftwindow(2),
makewindow(2,12,0,"Graph 2",0,0,18,35),
defineScale(2,0,500,0,300),
makeAxes(2,_,GWindow2,marker(50,d,3),marker(20,d,3),1,1,2,1),
element(carbody,E1),element(frontwindow,E2),
element(frontdoor,E3),element(rearwindow,E4),
element(light,E5),
shiftwindow(Gwindow2),
draw([d(1,E1),d(1,E2),d(1,E3),d(1,E4),d(1,E5)]),
wfs(_),
message("Text may be positioned according to scaled",
        "coordinates using scaleCursor:  For example:",
        "",
        "\tscaleCursor(100,40),write(\"Ford T - Year 1942\")",
        ""),
shiftwindow(Gwindow2),
scaleCursor(100,40),write("Ford T - Year 1942"),
wfs(_),
message("The following predicates use scaled coordinates:",
        "scaleCursor, scalePlot, and scaleLine. As an example",
        "of their use, we will shift back to the first graph",
        "and the corresponding coordinate system, and then",
        "draw a line. This can be performed by:",
        "\tshiftwindow(GWindow),shiftScale(1),scaleLine(0,0,100,0,1),"),
shiftwindow(GWindow),shiftScale(1),scaleLine(0,0,100,0,1),
wfs(_),
message("Any drawing or graph can be modified by",
        "changing the scale. In this example, it is used to",
        "make the car look more sporty.",
        "","\tdefineScale(3,0,400,0,400),",""),
defineScale(3,0,500,0,900),
modifyAxes(2,marker(50,d,3),marker(50,d,3)),
refreshAxes(2),
shiftwindow(Gwindow2),
draw([d(1,E1),d(1,E2),d(1,E3),d(1,E4),d(1,E5)]),
wfs(_),
message("Change the coordinate system to ZOOM in or out.",
        "Notice the effect when pressing either + or -." ,
        "(Any of the keys  l,r,u,d,<,>  will cause an effect).",
        "",
        "Use CTRL-BREAK to stop."),
    retract(newsline(_)),
    assertz(newsline("Press  +  -  <  >  l  r  u   or  d.    ")),
    repeat,
    wfs(C),
    write(C),
    process(C),
    refreshAxes(2),
    shiftwindow(Gwindow2),
    draw([d(1,E1),d(1,E2),d(1,E3),d(1,E4),d(1,E5)]),
   fail.

clauses
    element(carbody,
           [p(70,100),p(90,195),p(190,205),p(240,300),
            p(400,300),p(450,205),p(500,205),p(500,100),p(70,100)]).
    element(frontwindow,
           [p(243,290),p(320,290),p(320,210),p(200,210),p(243,290)]).
    element(frontdoor,
           [p(200,205),p(200,105),p(320,105),p(320,205),p(200,205)]).
    element(rearwindow,
           [p(330,290),p(398,290),p(438,210),p(330,210),p(330,290)]).
    element(light,[p(85,165),p(93,165),p(98,196)]).

    process('+') :- zoom(0,-100,0,-60).                                  /* Big */
    process('-') :- zoom(0,100,0,60).                                  /* Small */
    process('l') :- zoom(50,50,0,0).                                    /* Left */
    process('r') :- zoom(-50,-50,0,0).                                 /* Right */
    process('u') :- zoom(0,0,-50,-50).                                    /* Up */
    process('d') :- zoom(0,0,50,50).                                    /* Down */
    process('w') :- zoom(0,-50,0,0).                                    /* Wide */
    process('t') :- zoom(0,0,0,-50).                                    /* Tall */
    process('<') :-
          activescale(N),!,
          scale(N,X1,X2,_,_),!,
          L = X2-X1,H = X1-X2,
          zoom(L,H,0,0).
    process('>') :-
          activescale(N),!,
          scale(N,_,_,Y1,Y2),!,
          L = Y2-Y1,H = Y1-Y2,
          zoom(0,0,L,H).

    zoom(Dxl,Dxh,Dyl,Dyh) :-
          activescale(N),!,
          scale(N,Xmin,Xmax,Ymin,Ymax),!,
          NewXl = Xmin+Dxl,
          NewYl = Ymin+Dyl,
          NewXh = Xmax+Dxh,
          NewYh = Ymax+Dyh,
          NewXl<>NewXh,
          NewYl<>NewYh,!,
          retract(scale(N,_,_,_,_)),!,
          asserta(scale(N,NewXl,NewXh,NewYl,NewYh)).


message(S1,S2,S3,S4,S5):-
          shiftwindow(Old),shiftwindow(3),
          clearwindow,write(S1,"\n",S2,"\n",S3,"\n",S4,"\n",S5),shiftwindow(Old).
```

```
wfs(C) :-
      shiftwindow(4),clearwindow,
      newsline(S),field_str(0,0,20,S),
      keypressed,readchar(C),!,shiftwindow(3).
wfs(C) :- wait(3000),
      retract(newsline(String)),!,
      frontstr(1,String,F,Rest),
      concat(Rest,F,New),
      assertz(newsline(New)),!,
      wfs(C).

wait(0) :- !. wait(N) :- N1 = N-1,wait(N1).

function(N) :- N>100,!.
function(N) :- Y = 100*cos(6.28*N/50),scalePlot(N,Y,1),N1 = N+1,function(N1).
```

Other Tool Predicates for Handling Coordinates

This section explains predicates that set or convert coordinates.

Setting a Scale

The Toolbox predicate *findScale* evaluates an image, then automatically defines an appropriate scale so that the image fits into a given window. A call takes the following form:

```
findScale(ScaleNo,Drawing,Xfactor,Yfactor)
```

Xfactor indicates the amount of stretch (if any) you desire in the X direction when the images specified for this scale are represented on screen; *Yfactor* is the amount of stretch in the Y direction. Hence, the following call defines a scale that is twice as big on the X axis as the original DRAWING and is 1.3 times as big on the Y Axis. The identification number of the scale defined is returned in the parameter *ScaleNo*:

```
setscale(ScaleNo, [p(1,1),p(9,1),p(9,9),p(1,9)], 2, 1.3)
```

Conversion Between Virtual and Text Coordinates

The conversion predicate *virtual_text* is found in GPREDS.PRO. It enables conversion between virtual coordinates and character positions specified by a text row and column number. Its declaration takes the form

```
virtual_text(VROW,VCOL,ROW,COL)
```

When converting from a character position to virtual coordinates, what results is the coordinates of the upper left-hand corner of that character position. Thus,

```
virtual_text(R,C,0,0)
```

binds *R* and C to zero. The call (in 80-column mode)

```
virtual_text(R,C,24,79)
```

gives the virtual coordinates for the character in the lower right-hand corner of the entire screen; that is, (30720,31600). This enables the use of virtual coordinates based upon text coordinates but with their origin still at (0,0).

GPREDS.PRO also contains the tool predicate that is called in the form

```
gwrite(Row,Col,String,Color,Direction)
```

It writes the given *String* horizontally if *Direction* equals 0 or vertically if *Direction* equals 1. The first character is written in text position (*Row,Col*) using the indicated *Color*.

Conversion Between Scaled and Text or Virtual Coordinates

Conversion from or to scaled coordinates is handled by two tool predicates included in the file GGRAPH.PRO, declared as follows:

```
scale_virtual(X,Y,VROW,VCOL)
scale_text(X,Y,ROW,COL)
```

Conversions from scaled coordinates to either virtual or text coordinates are carried out relative to the currently active window. The full range for the scaled coordinates—found in the relevant scale definition of the form

```
scale(No,Xmin,Xmax,Ymin,Ymax)
```

—is assumed to fill the entire window, represented by *No.*

When converting from a character position to scaled coordinates, the outcome is the coordinates of the lower left-hand corner of that character position. This enables you to work with a coordinate system with origin (0,0) in the lower left-hand corner of the active window.

All the conversion predicates are independent of the actual screen mode. This is accomplished using two predicates: *getcols(c),* which binds C to the actual number of columns on the entire screen, and *getscreenmode(M),* which returns the current screen mode. Another useful related predicate is *screenarea(Rows,Cols),* which returns the number of rows and columns in the drawing area of the active window. These low-level tool files are supplied in GPREDS.PRO and GGRAPH.PRO.

Pie Charts

This section requires the file GPIE.PRO. Pie charts graphically show the percentages or parts that make up a whole "pie." The Toolbox provides the tool predicate *pieChart,* which draws a pie chart corresponding to the given input parameters, namely:

```
pieChart(vrow,vcol,vradius, pieSegmentList)
```

The arguments of this predicate represent the following:

vrow and *vcol*	the coordinates for the center of the pie
vradius	the radius of the pie

pieSegmentList[Slice(PercentageValue,label,fill,frame),Slice(...)]
a list of the "slices," each specified with:

percentage value: A negative percentage causes the slice to stick out from the rest of the pie.

optional label: If the label ends with an = character, the label is suffixed with the actual corresponding percentage value.

color-fill value: A 0 color value means only the frame of the slice is to be colored; a positive value means the entire slice is to be filled.

frame-color value: This specifies the color you want the frame and the label to be.

pieChart draws using virtual coordinates, that is, the center (specified as (*Row,Column*)) and the *Radius* are given in scaled coordinates. If the percentages given do not add up to 100 percent, *pieChart* scales up the slices specified so that they occupy the whole pie. This also allows the pie chart to be drawn in windows of different sizes.

The definition of *pieChart* is in the file GPIE.PRO. It can be modified and/or expanded. For example, you can make the slice labels look different. The pie slices are drawn by calling the external predicate *sector*, which you met in an earlier section and which is implemented in C. Thus, programs that use pie charts need to be compiled and linked as projects.

Using a Pie Chart in a Sample Program

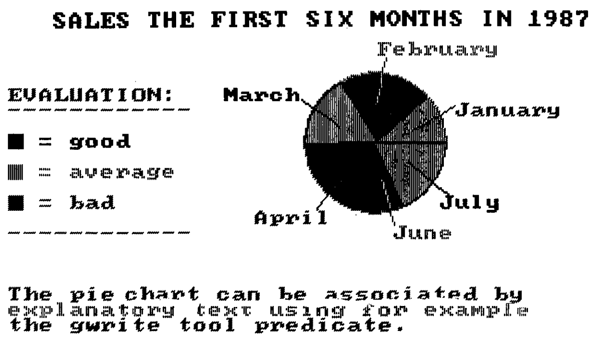
The following program produces the pie chart shown in Figure 4.3, which is shown right after the program listing.

XPIE.PRO

```
project "xpie"

include "tdoms.pro"
include "gdoms.pro"

database
   newsline(string)

include "gglobs.pro"
include "tpreds.pro"
include "gpreds.pro"
include "gpie.pro"

predicates
   message(string,string,string,string,string)
   wfs(char)
   wait(integer)
```

```
goal
   assertz(newsline("Press the space bar to continue.   ")),
   graphics(1,1,1),
   makewindow(1,6,0,"",0,0,18,40),
   makewindow(3,7,0,"",18,0,6,40),
   makewindow(4,8,0,"SPACE BAR MESSAGE",24,5,1,20),
   message("PIE CHARTS are easily drawn by giving",
           "the CENTER and RADIUS of the",
           "piechart and the PERCENTAGE,",
           "FRAMECOLOR, FILLCOLOR, and",
           "LABEL for each slice in the pie."),
   shiftwindow(1),
   pieChart(11000,19000,6000,
            [slice(10.6,"January",1,2),
             slice(20.7,"February",2,3),
             slice(15.1,"March",3,2),
             slice(23.5,"April",1,3),
             slice(5,"June",2,1),
             slice(17,"July",3,2)]),
   wfs(_),
   message("To attach an explanatory text to",
           "a pie chart, you can use",
           "the gwrite tool predicate.",
           "",
           ""),
   shiftwindow(1),
   gwrite(0,5,"SALES: THE FIRST SIX MONTHS IN 1987",3,0),
   gwrite(5,0,"EVALUATION:",1,0),
   gwrite(6,0,"------------",1,0),
   gwrite(8,0,"\219 = good",3,0),
   gwrite(10,0,"\219 = average",2,0),
   gwrite(12,0,"\219 = bad",1,0),
   gwrite(14,0,"------------",1,0),
   wfs(_),
   message("If certain slices need special",
           "attention, they can be moved outward",
           "and their labels suffixxed with",
           "a percentage.",
           ""),
   shiftwindow(1),
   clearwindow,
   pieChart(11000,19000,6000,
            [slice(10.6,"January",1,2),
             slice(-20.7,"February=",2,3),
             slice(15.1,"March",3,2),
             slice(-23.5,"April=",1,3),
             slice(5,"June",2,1),
             slice(17,"July",3,2)]),
   gwrite(0,5,"SALES: THE FIRST SIX MONTHS IN 1987",3,0),
   gwrite(4,0,"EVALUATION:",3,0),
   gwrite(6,0,"------------",1,0),
   gwrite(8,0,"\219 = good",3,0),
   gwrite(10,0,"\219 = average",2,0),
   gwrite(12,0,"\219 = bad",1,0),
   gwrite(14,0,"------------",1,0),
   wfs(_),
   message("Two small pie charts can be ",
           "used for comparing different periods.",
           "",
          "",
          ""),
  shiftwindow(1),
  clearwindow,
  pieChart(15000,7000,3000,
           [slice(10.6,"Jan",1,2),
            slice(20.7,"Feb",2,3),
            slice(15.1,"Mar",3,2),
            slice(17,"Apr",2,1)]),
  pieChart(15000,20000,3000,
           [slice(16,"May",1,2),
            slice(10.7,"Jun",2,3),
            slice(15.1,"Jul",3,2),
            slice(6,"Aug",2,1)]),
  wfs(_),
  message("The next demo requires an EGA card.",
          "16 colors are used to distinguish",
          "the different slices.",
          "Press E if the EGA card is attached;",
          "otherwise, press the space bar"),

  retract(newsline(_)),
  assertz(newsline("Press E when using EGA card  ")),
  wfs(Key),
  str_char(K,Key),
  upper_lower(K,"e"),
  graphics(4,1,1),
  makewindow(1,6,1,"",0,0,18,80),
  makewindow(3,7,1,"",18,0,6,80),
  makewindow(4,8,0,"SPACE BAR MESSAGE",24,30,1,20),
  message("Pie chart using EGA mode with 640 x 200 pixels in 16 colors",
          "",
          "",
          "",
          ""),
  shiftwindow(1),
  gwrite(0,0,"SALES: JANUARY 1986 - MARCH 1987",1,0),
  gwrite(3,0,"Colors of the two best months:",1,0),
  gwrite(5,10,"\219",7,0),
  gwrite(7,10,"\219",10,0),

  pieChart(11000,20000,6500,
           [slice(5,"Jan86",1,1),
            slice(7,"Feb86",2,2),
            slice(8,"Mar86",3,3),
            slice(5,"Apr86",4,4),
            slice(6,"May86",5,5),
            slice(6,"Jun86",6,6),
            slice(-9,"Jul86=",7,7),
            slice(4,"Aug86",8,8),
            slice(8,"Sep86",9,9),
            slice(-9,"Oct86=",10,10),
            slice(6,"Nov86",11,11),
            slice(3,"Dec86",12,12),
            slice(5,"Jan87",13,13),
            slice(7,"Feb87",14,14),
            slice(3,"Mar87",15,15)]),
  retract(newsline(_)),
  assertz(newsline("Press the space bar to continue. ")),
   wfs(_),
   graphics(5,1,1),
   makewindow(1,6,1,"",0,0,18,80),
   makewindow(3,7,1,"",18,0,6,80),
   makewindow(4,8,0,"SPACE BAR MESSAGE",24,5,1,20),
   message("EGA supports 640 x 350 pixel resolution in 16 colors.",
           "Higher resolution implies a decrease in speed.",
           "",
           "",
           ""),
   shiftwindow(1),
   gwrite(0,0,"SALES: JANUARY 1986 - MARCH 1987",1,0),
   gwrite(3,0,"The two best months:",1,0),
   gwrite(5,5,"Jul86  \219",7,0),
   gwrite(7,5,"Oct86  \219",10,0),

   pieChart(11000,20000,6000,
            [slice(5,"Jan86",1,1),
             slice(7,"Feb86",2,2),
             slice(8,"Mar86",3,3),
             slice(5,"Apr86",4,4),
             slice(6,"May86",5,5),
             slice(6,"Jun86",6,6),
             slice(-9,"Jul86=",7,7),
             slice(4,"Aug86",8,8),
             slice(8,"Sep86",9,9),
             slice(-9,"Oct86=",10,10),
             slice(6,"Nov86",11,11),
             slice(3,"Dec86",12,12),
             slice(5,"Jan87",13,13),
             slice(7,"Feb87",14,14),
             slice(3,"Mar87",15,15)]),
   wfs(_).

clauses
   message(S1,S2,S3,S4,S5) :-
         shiftwindow(Old),shiftwindow(3),
         clearwindow,
         attribute(1),write(S1,"\n"),
         attribute(2),write(S2,"\n"),
         attribute(1),write(S3,"\n"),
         attribute(2),write(S4,"\n"),
         attribute(1),write(S5,"\n"),
         shiftwindow(Old).

   wfs(C) :-
         shiftwindow(4),clearwindow,
         attribute(1),
         newsline(S),field_str(0,0,20,S),
         keypressed,readchar(C),!,shiftwindow(3).

   wfs(C) :- wait(8000),
         retract(newsline(String)),!,
         frontstr(1,String,F,Rest),
         concat(Rest,F,New),
         assertz(newsline(New)),
         wfs(C).
```

```
wait(0) :- !. wait(N) :- N1 = N-1,wait(N1).
```



The pie chart can be associated by explanatory text using for example the gwrite tool predicate.

Figure 4.3: Sales: The First Six Months in 1987

Bar Charts

This section uses the file GBAR.PRO. Bar charts are another way of representing sets of numerical data. You generate a bar chart by calling one of these tool predicates:

```
bargraph(Left,Bottom,Right,Top,BarRatio,Barlist,Factor)
bargraph3d(Left,Bottom,Right,Top,BarRatio,Theta,Barlist,Factor)
```

Either predicate can generate colored bar charts; the *bargraph3d* bar chart resembles three-dimensional boxes. The position of the bar chart relative to the borders of the currently active window is specified on a character basis by the four parameters *Left, Bottom, Right,* and *Top.*

For a three-dimensional bar chart, the viewing angle is passed as an angle measured in radians. The parameter BarRatio specifies the width of the bars relative to the spacing between the bars. A spacing of 0.5 indicates that the bars have the same width as the gaps between them.

The list of bars belongs to the domain

```
BARLIST = BAR*
BAR = bar(VHEIGHT,STRING,COLOR,COLOR) ; space
```

A bar is usually specified by its height, a label (which can be an empty string), a frame color, and a fill color. A bar can also be a *space*, placed there only to separate the bars visually. The bars are automatically scaled to fill the specified area of the active window. The scaling factor is returned as the last parameter.

Using *bargraph* and *bargraph3d* in a Sample Program

This program is called XBAR.PRO. As the bar-chart tool predicates use the *box* predicate, which is defined as an external predicate implemented in C, this demo must be compiled as an project. The corresponding module list, XBAR.PRJ, should therefore contain *graphics+xbar+*.

XBAR.PRO

```
project "xbar"

include "tdoms.pro"
include "gdoms.pro"
include "gglobs.pro"
include "tpreds.pro"
include "gpreds.pro"
include "gbar.pro"

goal
   Theta = 0.3,
   Graph1Ratio = 0.5,
   graphics(1,1,1),
   gwrite(0,12,"3-D BAR CHART.",3,0),
   gwrite(1,12,"--------------",2,0),
   gwrite(2,1,"SALES IN BILLION $",1,1),
   gwrite(24,0,"Resolution: 320x200 pixels in 4 colors",1,0),
   BarGraph3D(3,4,4,4,Theta,Graph1Ratio,
              [bar(2,"",1,2),
               bar(3,"1984",1,3),
               bar(5,"",1,2),
               space,
               bar(4,"",1,2),
               bar(7,"1985",1,3),
               bar(7,"",1,2),
               space,
               bar(4,"",1,2),
               bar(7,"1986",1,3),
            bar(7,"",1,2),
            space,
            space,
            space,
            bar(10,"Estim. 1987",1,3)],_),
readchar(_),
graphics(2,1,1),
makewindow(1,1,0,"",0,0,25,80),
gwrite(0,30,"2-D BAR CHART",1,0),
gwrite(1,30,"-------------",1,0),
Graph2Ratio = 0.9,
gwrite(24,0,"Resolution: 640x200 in 2 colors.",1,0),
BarGraph(5,5,4,4,Graph2Ratio,
         [bar(52,"",1,0),
          bar(33,"1983",1,1),
          bar(70,"",1,0),
          space,
          bar(49,"",1,1),
          bar(80,"1984",1,0),
          bar(100,"",1,1),
          space,
          bar(40,"",1,0),
          bar(50,"1985",1,1),
          bar(70,"",1,0),
          space,
          bar(100,"Reference",1,1)],SFactor),

gwrite(22,0,"The graph is automatically scaled to fit a specified area.",1,0),
str_real(ScaleStr,SFactor),
concat("The scaling factor in this example is =",ScaleStr,Str1),
gwrite(23,0,Str1,1,0),
readchar(_),
graphics(4,1,1),
makewindow(4,7,0,"",0,0,25,80),
Graph3Ratio = 0.9,
gwrite(24,0,"Resolution: EGA 640 x 200 in 16 Colors",1,0),
BarGraph3D(7,7,7,7,Theta,Graph3Ratio,
         [bar(2,"",1,2),
          bar(3,"1983",3,4),
          bar(5,"",5,6),
          space,
          bar(4,"",7,8),
          bar(7,"1984",9,10),
          bar(7,"",11,12),
          space,
          bar(4,"",13,14),
          bar(7,"1985",15,1),
          bar(7,"",2,3),
          space,
          bar(10,"Reference",4,5)],_),
readchar(_),
graphics(5,1,1),
makewindow(4,7,0,"",0,0,25,80),    Graph3Ratio = 0.9,
gwrite(24,0,"Resolution: EGA 640 x 350 in 16 Colors",1,0),
BarGraph3D(7,7,7,7,Theta,Graph3Ratio,
         [bar(2,"",1,2),
          bar(3,"1983",3,4),
          bar(5,"",5,6),
          space,
            bar(4,"",7,8),
            bar(7,"1984",9,10),
            bar(7,"",11,12),
            space,
            bar(4,"",13,14),
            bar(7,"1985",15,1),
            bar(7,"",2,3),
            space,
            bar(10,"Reference",4,5)],_),
    readchar(_).
```

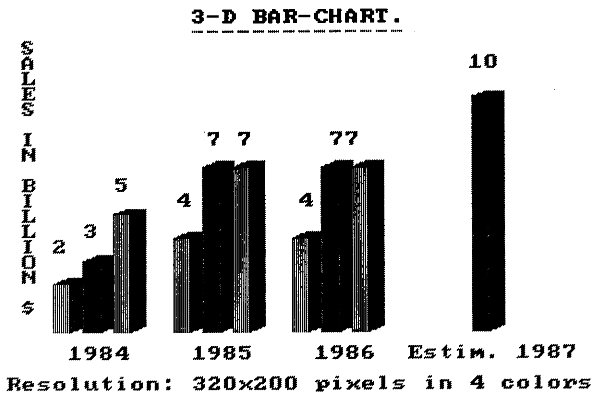


Figure 4.4: Three-Dimensional Bar Chart

Color Graphics with the EGA Card

This section refers to the file GEGA.PRO. The Turbo Prolog standard predicate *graphics* prepares the screen for graphics, since using the screen for graphics display differs from using it for text. In graphics, each pixel on the screen is represented by 1 or 2 *bits* in graphics modes 1 and 2 (the CGA modes) and 4 bits in graphics mode 3, 4, and 5 (the EGA modes). In text, however, each character is represented by two *bytes*, one giving the ASCII code for that character and the other the attribute with which that character is to be displayed.

In EGA graphics mode, the 4 bits representing each pixel allow you a choice of 16 colors. One color is the background color—represented as color

0—so this leaves 15 colors to use when drawing and writing in graphics mode.

The choice of colors is made using the *attribute* standard predicate, whether in text or graphics mode. Thus,

```
attribute(1),write("This is written using color 1")
```

works in both text and graphics mode.

The *graphics* predicate takes three parameters:

```
Mode,Palette,BackGround
```

The choices of *Palette* have meaning only in CGA graphics mode 1 (see the *Turbo Prolog Owner's Handbook*).

The EGA adapter supports a palette with programmable colors. Two tool predicates may be used to manipulate these:

```
setEGApalette(ColorList)
setEGAregister(Register,Color)
```

They are found in the file GEGA.PRO, and the following example demonstrates how they work.

Using the EGA Palettes in a Sample Program

The sample program in the file XEGA.PRO enables its user to set up the colors of an EGA screen and save the corresponding list of (17) integers in the file COLORS.DEF. You may want to experiment by building an application that allows the user to switch between several color definitions.

XEGA.PRO

```
include "tdoms.pro"
include "gdoms.pro"

domains
   FILE = datafile

database
   color(integer,integer)
   actualcolor(integer)
```

```
include "tpreds.pro"
include "gpreds.pro"
include "gega.pro"

predicates
   test(integer)
   choice(key)
   changecolor(integer)
   shiftcolor(color)
   getColorList(integerlist)
   showfield(integer)
   loop
clauses
/* Default settings of the EGA palette registers */
   color(0,0).
   color(1,1).
   color(2,2).
   color(3,3).
   color(4,4).
   color(5,5).
   color(6,6).
   color(7,7).
   color(8,56).
   color(9,57).
   color(10,58).
   color(11,59).
   color(12,60).
   color(13,61).
   color(14,62).
   color(15,63).
   color(16,0).

   test(17) :- !.
   test(N) :-
         str_int(S,N),
         concat("Color ",S,Sn),
         FieldRow = N+3,N_ = N,
         gwrite(FieldRow,20,Sn,N_,0),
         N1 = N+1,
         test(N1).

   choice(up) :- !,
         shiftcolor(-1).
   choice(down) :- !,
         shiftcolor(1).
   choice(left) :- !,
         changecolor(-1).
   choice(right) :- !,
         changecolor(1).
   choice(char('s')) :- !,
         getColorList(L),
         openwrite(datafile,"color.def"),
         writedevice(datafile),
         write(L),
         closefile(datafile),
         writedevice(screen).
   choice(_) :-
         beep.
   shiftcolor(N) :-
         retract(actualcolor(C)),!,
         NewC = abs(C+N+17) mod 17,
         assertz(actualcolor(NewC)),!.

   changecolor(N) :-
         actualcolor(AC),
         retract(color(AC,Color)),!,
         NewColor = abs(Color+N+64) mod 64,
         assertz(color(AC,NewColor)),!.

   getColorList([Bgrnd,C1,C2,C3,C4,C5,C6,C7,C8,C9,C10,C11,C12,C13,C14,C15,
                 Border]) :-
         color(0,Bgrnd),
         color(1,C1),         color(2,C2),      color(3,C3),      color(4,C4),
         color(5,C5),         color(6,C6),      color(7,C7),      color(8,C8),
         color(9,C9),         color(10,C10),    color(11,C11),    color(12,C12),
         color(13,C13),       color(14,C14),    color(15,C15),
         color(16,Border),!.

   showfield(0) :-
         color(0,Color),!,
         attribute(1),write(" (Background) =",Color).
   showfield(16) :-
         color(16,COlor),!,
         attribute(Color),write(" (Border) =",Color).
   showfield(N) :-
         color(N,COlor),!,
         attribute(Color),
         write(" ---> ",Color).

loop :-
         actualcolor(OldAct),!,
         FieldRow = OldAct+3,
         cursor(FieldRow,40),
         showfield(OldAct),
         readkey(K),
         choice(K),
         actualcolor(Regno),!,
         color(Regno,Color),!,
         setEGAregister(Regno,Color),
         cursor(FieldRow,40),
         write("                    "),
         fail.

goal
   graphics(5,1,1),
   gwrite(0,15,"DEFINITION OF THE 17 EGA-PALETTE REGISTERS",1,0),
   gwrite(2,20,"Register (0-16)",1,0),
   gwrite(2,40,"Color code (0-63)",1,0),
   assertz(actualcolor(1)),
   getcolorlist(L),
   setEGApalette(L),
   test(1),
   gwrite(3,20,"Background",1,0),
   gwrite(19,20,"Border",1,0),
   cursor(21,2),
   attribute(1),write(" Use "),attribute(15),write("\24"),
   attribute(1),write(" or "),attribute(15),write("\25"),
attribute(1),write(" cursor keys to select registers and "),
attribute(15),write("\26"),
attribute(1),write(" or "),
attribute(15),write("\27"),
attribute(1),write(" to change color."),
cursor(22,2),write("Press "),attribute(15),write(s),
attribute(1),write("To save the defining 17 integers in the file
            \"color.def\"."),
line(2400,0,2400,31999,1),
line(2400,0,31999,0,1),
line(31999,0,31999,31999,1),
line(31999,31999,2400,31999,1),
repeat,
loop.
```

Loading .PIC Files

In defining a call to loadpic, Filename should be FileName

The Turbo Prolog tool predicates *loadpic* and *savepic* are supplied in the .OBJ file PICTOOLS.OBJ, created from a program written in C. Any program that uses them must therefore be incorporated into a *project* file, so that the program can be linked with the .OBJ file containing the tools. The screen should be in mode 1.

It also follows that *loadpic* and *savepic* must be declared in the global predicates section of any program that uses them. The global declarations belonging to the graphics tools are collected in the file GGLOBS.PRO. In this file, the declarations of *loadpic* and *savepic* take the following form:

```
global predicates

   loadpic(String,Integer,Integer,Integer,Integer,Integer,Integer)
          - (i,i,i,i,i,i,i) language c

   savepic(String) - (i) language c
```

A call to *loadpic* of the form

```
 loadpic(FileName, StartRowPicFile, StartColPicFile,
        StartRowScreen, StartColScreen, NoOfRows, NoOfCols)
```

in which all the parameters are bound, loads a full screen graphics image from the file determined by *Filename*. For the purpose of addressing various portions of this image, it is regarded as occupying 200 rows, numbered 0 through 199, and 320 columns, numbered 0 through 319. *StartRowPicFile* and *StartColPicFile* specify the top left-hand corner of the sub-image to be selected from the full screen image. *NoOfRows* and *NoOfCols* complete the specification of this sub-image. *StartRowScreen* and *StartColScreen* specify where on the screen the top left-hand corner of the sub-image is to be displayed.

Using *loadpic* in a Sample Program

The following sample program (from XPICDEMO.PRO) demonstrates the facilities provided by *loadpic*. It uses the two .PIC files, WELCOME.PIC and TEST.PIC. For each image in these files, the program first displays the whole image and then various parts of that image in various positions on the screen. After each image or sub-image has been displayed, the program waits for the user to press the space bar before continuing.

XPICDEMO.PRO

```
project "xpicdemo"

include "tdoms.pro"
include "gglobs.pro"
include "tpreds.pro"
include "menu.pro"

predicates
   picture(STRING)

goal
   picture(X),
   graphics(1,1,2),
   loadpic(X,0,0,0,0,199,319),readchar(_),
   graphics(1,1,3),
   write("\nPIC 1: Whole screen"),readchar(_),
   loadpic(X,0,0, 0,0,199,319),readchar(_),clearwindow,
   write("\nPIC 2: Upper Left-Hand Corner "),readchar(_),
   loadpic(X,0,0, 0,0,100,160),readchar(_),clearwindow,
   write("\nPIC 3: Bottom Left-Hand Corner "),readchar(_),
   loadpic(X,100,0, 100,0,100,160),readchar(_),clearwindow,
   write("\nPIC 4: Bottom Right-Hand Corner "),readchar(_),
   loadpic(X,100,160,100,160,100,160),readchar(_),clearwindow,
   write("\nPIC 5: Upper Right-Hand Corner "),readchar(_),
   loadpic(X,0,160, 0,160,100,160),readchar(_),clearwindow,
   write("\nPIC 6: Top Left Corner ==> Bottom Right Corner "),readchar(_),
   loadpic(X,0,0,100,160,100,160),readchar(_),clearwindow,
   write("\nPIC 7: Middle ===> Upper Left-Hand Corner "),readchar(_),
   loadpic(X,50,80, 0,0,100,160),readchar(_),clearwindow,fail.
clauses
   picture("Welcome.pic").
   picture("Test.pic").
```

loadpic's companion, *savepic,* saves a whole screen image in a file. Its declaration was shown earlier in this section. A call of the form

```
savepic(Filename)
```

stores the currently selected graphics screen image in the file specified by *Filename.*

Using *loadpic* and *savepic* to Create an On-Screen Presentation

In this section, the *loadpic* and *savepic* predicates are used to construct a program that saves each of several screen displays of large text letters in .PIC files and then runs through them like a slide show. Thus, you can use this program to construct a set of notes to accompany a lecture. In fact, the program allows the production of any number of slides, each slide being saved in a disk file when the # key is pressed. Esc is pressed when the specification of slides is complete.

A simple menu offers the choice between making slides and viewing a presentation. If you choose to view a presentation, the slides are shown in the order they were made in, with each slide remaining on the screen until the Space bar is pressed.

The program uses the screen image in TEST.PIC, which contains an alphabet written in large (lowercase) letters. *loadpic* is used to select one of the letters in this full-screen image so that the PC keyboard can be used like a typewriter (lowercase letters only) during slide creation. The position (*Row* and *Column*) of the top left-hand corner of each of the letters is held in the predicate *index*. For example, the clause

```
index('p',84,94)
```

indicates that the top left-hand corner of the letter *p* begins at row 84, column 94 of the full-screen image.

getnewpositions controls how letters you type are displayed on the screen. *findpicture* helps to take special care of the Space bar and Return, both of which affect the positioning of letters on the display screen but do not produce images on a slide.

XSLIDES.PRO

```
project "xslides"

include "tdoms.pro"

database
   index(CHAR,INTEGER,INTEGER)
   slide(INTEGER,STRING)

include "gglobs.pro"
include "preds.pro"
include "menu.pro"

predicates
   mydisplay(ROW,COL,INTEGER)
   findpicture(char,ROW,COL,ROW,COL,INTEGER,INTEGER)
   getnewpositions(ROW,COL,ROW,COL)
   decide(INTEGER)

goal
   makewindow(11,7,0,"",0,0,25,80),
   repeat,text,
   menu(10,10,64,23,["Make some slides","See the slide show","Exit"],
        "Choose an option",1,Choice),
   decide(Choice),text,
   fail.

clauses
   decide(1) :-
         graphics(1,1,2),
         attribute(2),
         mydisplay(0,0,0).

   decide(2) :-
         slide(_,B),graphics(1,1,2),
         loadpic(B,0,0, 0,0,199,319),
         readchar(_),
         clearwindow,
         fail.

   decide(3) :- exit.

   mydisplay(Row,Col,Slideno) :-
         readchar(X),X<>'\027',
         findpicture(X,Row,Col,NewX,NewY,Slideno,NewSlideno),
         mydisplay(NewX,NewY,NewSlideno).

   findpicture('\27',_,_,_,_,_,_) :- !,fail.

   findpicture('#',_,_,0,0,Slideno,NewSlideno) :- !,
         str_int(SN,Slideno),
         concat("Slide",SN,FSN),
         assertz(slide(Slideno,FSN)),
         savepic(FSN),
          graphics(1,1,2),
          NewSlideno = Slideno+1.

     findpicture('',CurrScrRow,CurrScrCol,NewScrRow,NewScrCol,S,S) :- !,
          getnewpositions(CurrScrRow,CurrScrCol,NewScrRow,NewScrCol).

     findpicture('\13',CurrScrRow,_,NewScrRow,0,S,S) :- !,
          NewScrRow = CurrScrRow+25.

     findpicture(X,CurrScrRow,CurrScrCol,NewScrRow,NewScrCol,S,S) :-
          index(X,Row1,Col1), Row = Row1, Col = Col1,
          loadPic("test.pic",Row,Col,CurrScrRow,CurrScrCol,25,25),
          getnewpositions(CurrScrRow,CurrScrCol,NewScrRow,NewScrCol).

     getnewpositions(X,Y,NewX,0) :- Y+34> = 300,NewX = X+25.
     getnewpositions(X,_,0,0) :- X+25> = 199.
     getnewpositions(X,Y,X,NewY) :- NewY = Y+34.

     index('a',28,64).       index('b',25,94).       index('c',28,128).
     index('d',25,158).      index('e',28,188).      index('f',24,224).
     index('g',26,252).      index('h',54,64).       index('i',52,94).
     index('j',53,127).      index('k',56,158).      index('l',56,188).
     index('m',56,224).      index('n',56,252).      index('o',84,64).
     index('p',84,96).       index('q',84,126).      index('r',84,158).
     index('s',84,192).      index('t',84,224).      index('u',84,256).
     index('v',112,62).      index('w',112,94).      index('x',112,128).
     index('y',112,160).     index('z',112,192).
```

C H A P T E R

5

Communication with Remote Serial Devices

This chapter explains the communication tool predicates that the Toolbox offers. The first section deals with general serial communications, followed by several sample programs—including a subset of the popular XMODEM file-transfer protocol. Next, modem communication is discussed, along with the tools provided for this by the Toolbox. Finally, a complete menu-driven serial communications program is shown.

Before going on, note that you should be familiar with serial communications in general and the IBM-PC communication hardware in particular, especially the Asynchronous Adapter. More details can be found in the IBM-PC technical reference manual. Similarly, if you intend to use the communication tool predicates to communicate via a modem, study the *owner's manual* that comes with your modem.

The Communication Tool Predicates

The communication tools provided in the Turbo Prolog Toolbox are a powerful set of predicates that can be used in many different ways, including the following:

- to connect the serial port of your PC to that of another PC, so you can send and receive information either via the simple file-transfer program provided or via your own enhanced (full error-checking and error-correcting) version
- to construct a terminal-emulation program, so your PC can talk to a mini- or mainframe computer
- to connect your PC to a Hayes-compatible modem and communicate with other machines using a phone line

The communication tool predicates are buffered and fully interrupt-driven using the PC's interrupt controller. They work on either of the PC's serial ports: COM1, which is located at I/O-address $03F8; or COM2, which is located at I/O-address $02F8. Transmission rates can be freely selected among the most commonly used baud rates, in the range 110 to 9600 baud.

The Toolbox gives you two types of communication tool predicates: those you use with modems, and those you implement without modems. Those tool predicates not related to the use of modems are in the file SERIAL.OBJ, and those that work with modems are in the file MODEM.OBJ. The programs that make use of these communication tool predicates must be declared as part of a project and compiled as a project. (If you're not familiar with modular programming, see Appendix A, "Compiling a Project.") The accompanying global declarations are in the file COMGLOBS.PRO, which must be **included** in all programs that use the communications tools.

Hardware Considerations

When connecting a PC to a remote device, you must determine whether each device is configured as DTE (Data Terminal Equipment) or DCE (Data Communications Equipment). Since the IBM PC (and most IBM PC-compatibles) emulates DTE, it is only necessary to determine what signals the remote device emulates. This information can usually be found in the manual for that device. Once you've determined this, you can decide which of the following pinouts is required.

The following list shows commonly used communications terms and their meanings:

TX	= Transmitted data
RX	= Received data
DSR	= Data set ready
DTR	= Data terminal ready
Ground	= Signal ground
RTS	= Request to send
CTS	= Clear to send

Pinouts for DTE-to-DTE Configuration

The following pinout can generally be used when connecting two Pcs together directly (hard-wired). A cable with this configuration is known as a null modem cable.

IBM PC	RS232 ADAPTER		REMOTE DEVICE
TX	Pin 2	————————>	Pin 3 RX
RX	Pin 3	————————>	Pin 2 TX
DSR	Pin 6	————————>	Pin 20 DTR
Ground	Pin 7	————————>	Pin 7 Ground
DTR	Pin 20	————————>	Pin 6 DSR

In addition, to implement the RTS/CTS (Request To Send/Clear To Send) handshaking protocol, the following connections must be made:

RTS	Pin 4	————————>	Pin 5 CTS
CTS	Pin 5	————————>	Pin 4 RTS

Pinouts for DTE-to-DCE Configuration

The following pinout is most commonly used when connecting a PC to a modem. Be sure to check the owner's manual for the modem to see if it emulates DCE. A cable with this configuration is known as a straight-thru cable.

IBM PC	RS232 ADAPTER		REMOTE DEVICE
TX	Pin 2	——————>	Pin 2 RX
RX	Pin 3	——————>	Pin 3 TX
DSR	Pin 6	——————>	Pin 6 DTR
Ground	Pin 7	——————>	Pin 7 Ground
DTR	Pin 20	——————>	Pin 20 DSR

Make the following connections for RTS/CTS handshaking:

RTS	Pin 4	——————>	Pin 4 CTS
CTS	Pin 5	——————>	Pin 5 RTS

General Serial Communications

Serial communications simply means transferring one data bit at a time over a single wire. In this section, you'll learn how to open and close a serial port, perform transmission procedures and queue sizing, and delete buffers.

Opening a Serial Port

The tool predicate *openRS232* initializes either COM1 or COM2 so it is ready to transmit or receive data. *openRS232* will fail if the Asynchronous Adapter (or equivalent) is not in the PC, or if one of its parameters is out of range (for example, if an illegal value is given for the baud rate or I/O port number). In a containing program, *openRS232* must be declared as follows:

```
DetermOpenRS232(Integer,Integer,Integer,Integer,Integer,Integer,
                Integer,Integer) - (i,i,i,i,i,i,i,i) language c
```

(This is done automatically if you **include** COMGLOBS.PRO in the program.)

In a call of the form

```
openRS232(PortNo, InputBufSize, OutputBufSize, BaudRate,
          Parity, WordLength, StopBits, Protocol)
```

all the parameters must be bound and the following values filled in.

PortNo	= 1 means use COM1. = 2 means use COM2.
InputBufSize	must be in the range 1 to 16383 and specifies the number of bytes reserved for the input buffer.
OutputBufSize	must be in the range 1 to 32767 and specifies the number of bytes reserved for the output buffer.
BaudRate	is determined according to the following: = 0 means 110 Baud. = 1 means 150 Baud. = 2 means 300 Baud. = 3 means 600 Baud. = 4 means 1200 Baud. = 5 means 2400 Baud. = 6 means 4800 Baud. = 7 means 9600 Baud.
Parity	= 0 means no parity. = 1 means odd parity. = 2 means even parity.
WordLength	= 0 means 5 data bits. = 1 means 6 data bits. = 2 means 7 data bits = 3 means 8 data bits.
StopBits	= 0 means 1 stop bit. = 1 means 2 stop bits.
Protocol	= 0 means communication with no protocol (that is, *neither* XON/XOFF *nor* RTS/CTS). = 1 means communication *with* XON/XOFF but *without* RTS/CTS (our preferred mode). = 2 means communication *with* RTS/CTS but *without* XON/XOFF. If RTS (Request To Send) is high, then CTS will go high when the external device is ready to receive (and vice-versa). = 3 means communication with *either* XON/XOFF *or* RTS/CTS.

For instance, a Hewlett-Packard (HP) LaserJet printer requires the following: a transmission speed of 9600 baud; a data format consisting of no parity, eight data bits, and one stop bit; and the RTS/CTS protocol. Thus, a call of the form

```
  openRS232(1, 256, 256, 7, 0, 3, 0, 2)
```

initializes the COM1 port for printing using an HP LaserJet printer with input and output buffers of 256 bytes.

Closing a Serial Port

The tool predicate *closeRS232* closes an open communication port. This means the PC interrupt mechanisms are restored to the state they were in before the corresponding *openRS232* was executed and the input/output buffers de-allocated. It is extremely important to close a communication port before an application terminates because the interrupt routines redirect interrupts IRQ3 and IRQ4 from the interrupt controller. *closeRS232* fails if the communication port referred to isn't open or doesn't exist.

closeRS232 is declared in a containing program by a declaration of the form

```
  Determ closeRS232(Integer) - (i) language c
```

A call

```
  closeRS232(PortNo)
```

affects COM1 if *PortNo* is bound to 1 and COM2 if *PortNo* is bound to 2. Thus, *closePort* defined by

```
  closePort(PortNo) :-
       closeRS232(PortNo),!.
  closePort(PortNo) :-
       write("\nPort COM",PortNo, cannot be closed"),readchar(_).
```

either closes a serial I/O port or displays an error message if an attempt is made to close an unopened port.

Obtaining Transmission Status Information

The tool predicate *status_RS232* returns information concerning the current state of transmission. This information can be used during debugging, for example, or in the production of an error-checking and error-correcting file-transfer package.

status_RS232 returns a status value that is a bit mask, so that it is often necessary to use the ***bitand*** standard predicate to de-mask the value. The status value is reset before each write and read operation, and it is good practice to check the transmission status after each transmission. *status_RS232* fails if the specified I/O port has not been opened.

It is declared as follows:

```
Determ status_RS232(Integer,Integer) - (i,o) language c
```

A call of the form

```
status_RS232(PortNo,Status)
```

in which *PortNo* is bound to the code for an opened I/O port (*PortNo=1* means COM1, *PortNo=2* means COM2) binds *Status* to the bit mask value representing the current transmission status as shown:

Status	= 0	Transmission ok.
	= 1	Input characters have been lost because the input queue was full when characters were received.
	= 2	Parity error detected.
	= 4	Over-run detected.
	= 8	Framing error detected.
	= 16	Break signal detected.
	= 32	An XOFF has been received.
	= 64	An XON has been received.
	= 128	An XOFF has been transmitted.
	= 256	An XON has been transmitted.
	= 512	Input buffer is empty when trying to read.
	= 1024	Output buffer is full when trying to write.

Thus, the predicate *check_status* is defined by the following and can be used to display transmission status messages:

```
check_status :- status_RS232(1,Status),
     check_stat(Status,1, "Input Characters have been lost"),
     check_stat(Status,2, "Parity error"),
     check_stat(Status,4, "Overrun detected"),
     check_stat(Status,8, "Framing error detected"),
     check_stat(Status,16, "Break signal detected"),
     check_stat(Status,32, "An XOFF has been received"),
     check_stat(Status,64, "An XON has been received"),
     check_stat(Status,128, "An XOFF has been transmitted"),
     check_stat(Status,256, "An XON has been transmitted"),
     check_stat(Status,512, "Input buffer empty"),
     check_stat(Status,1024,"Output buffer full").

check_stat(Status,BitMask,Mess) :-
     bitand(Status,BitMask,V), V<>0,!,nl,write(Mess).

check_stat(_,_,_).
```

Transmit a Character from a Serial Port

The tool predicate *txCh_RS232* places a character in the output buffer if the buffer is not full. That character is then transmitted when the receiver is ready. Since the low-level transmission is interrupt driven, it is fully transparent when viewed from a Turbo Prolog program. A program is not normally aware of when transmission from the buffer takes place. However, the status of a transmission can always be monitored by calling the predicate *status_RS232*. *txCh_RS232* fails if the output buffer is full or the specified communication port is not open. It must be declared in any containing program by means of a declaration of the form

```
Determ txCh_RS232(Integer,Char) - (i,i) language c
```

Then a call of the form

```
txCh_RS232(PortNo,CH)
```

transmits the character *CH* to the output buffer for port number *PortNo*, where *PortNo=1* means the COM1 serial communication port and *PortNo=2* means the COM2 serial communication port.

To monitor the transmission state after a character has been transmitted, use the predicate *send_ch*, defined as follows:

```
send_ch(CH) :- txCh_RS232(1,CH),!.
send_ch(_) :-  write("Error sending character "),
     status_RS232(1,Status),
     write("Status code=",Status).
```

Receiving a Character from a Serial Port

The tool predicate *rxch_RS232* returns any characters from the input buffer, according to this declaration:

```
Determ rxch_RS232(Integer,Char) - (i,o) language c
```

So that a call of the form

```
rxch_RS232(PortNo,CH)
```

binds *CH* to the next available character (if any) from the input buffer for port number *PortNo*, where *PortNo=1* means the COM1 communication port and *PortNo=2* means the COM2 communication port.

rxch_RS232 fails if the input buffer is empty or the specified port is not open. If anything goes wrong, you can obtain more information about the current transmission state by calling the *status_RS232* tool predicate as described earlier.

Obtaining Input and Output Queue Sizes

The tool predicate *queuesize_RS232* returns the size of the input and output queues. These, respectively, contain the characters that the low-level routines have received but have not been read yet by the calling Turbo Prolog program and the characters the Turbo Prolog program has written but have not been transmitted yet. *queuesize_RS232* is declared as follows:

```
Determ queuesize_RS232(Integer, Integer, Integer) - (i,o,o) language c
```

In a call of the form

```
queuesize_RS232(PortNo, SizeOfInputQueue, SizeOfOutputQueue)
```

with *PortNo* bound to the code for a serial I/O port (*PortNo=1* means COM1, *PortNo=2* means COM2), *SizeOfInputQueue* and *SizeOfOutputQueue* become bound, respectively, to the number of characters in the input and output queues. *queuesize_RS232* fails if the specified COM port is not open. Thus, the clause *test_queue* defined by

```
test_queue :- queuesize_RS232(1,SizeI,SizeO),
      write("No of characters in input queue: ",SizeI),
      write("No of characters in output queue: ",SizeO).
```

displays the sizes of the input and output queues on the screen.

Deleting the Output Buffer

delOutBuf_RS232 deletes the contents of the output queue. This predicate is useful in circumstances where it is necessary to retransmit a certain block of data. It fails if the specified port is not open. *delOutBuf_RS232* is declared as

```
Determ delOutBuf_RS232(Integer) - (i) language c
```

so that a call of the form

```
delOutBuf_RS232(PortNo)
```

with *PortNo* bound to the code for an I/O port, deletes the contents of the output buffer for that port.

Deleting the Input Buffer

delInBuf_RS232 deletes the contents of the input queue. It is useful in cases where data input during a transmission phase should be suppressed. *delInBuf_RS232* fails if the specified port is not open. It is declared as follows:

```
Determ delInBuf_RS232(Integer) - (i) language c
```

In a call of the form

```
delInBuf_RS232(PortNo)
```

with *PortNo* bound to the code for an I/O port, deletes the contents of the input buffer for that port. The call

```
delInBuf_RS232(1)
```

deletes the contents of the buffer for COM1.

Some Complete Sample Programs

Following are four sample communication programs. They include a printer driver, a terminal emulator, polled communication, and an XMODEM subset.

A Printer Driver That Lets You Modify the File Being Printed

The following sample program, XPRINTER.PRO, reads the contents of the text file DATA.TRS, converts those characters to uppercase, and then sends the result to an HP LaserJet printer (or compatible) connected via COM1. It can be modified for use with your own serial printer by changing the parameters in *openRS232(...)*.

The program contains a "time out" feature. If you are unable to transmit a character for any reason, the transmission is attempted for 5 seconds, after which time the transmission status is displayed and transmission halted. To time this, the program uses the *ticks* tool predicate, which is contained in both SERIAL.OBJ and MODEM.OBJ. It suspends the current program for a certain amount of time and is declared as

```
ticks(Integer)
```

A call of the form

```
ticks(TimeInHundredthsOfSec)
```

with *TimeInHundredthsOfSec* bound to 50, for example, suspends the current program for half a second.

XPRINTER.PRO

```
project "xprinter"

domains
   FILE = sp

include "comglobs.pro"

predicates
   run
   process_until_eof
   openfile(file,string)
   transmit(char)
   trans_ch_delay(char,integer)
   wait_until_empty_output_buffer(integer)

goal
   run.

clauses
   run :-
         PortNo = 1,                                              /* COM1 */
         InputBufSize = 1,                                        /* Size of input buffer */
         OutputBufSize = 4000,                                    /* Size of output buffer */
         BaudRate = 7,                                            /* 9600 bits per second */
         Parity = 0,                                              /* No parity */
         WordLength = 3,                                          /* Eight data bits */
         StopBits = 0,                                            /* One stop bits */
         Protocol = 2,                                            /* XON/XOFF can not be used */
         openRS232(PortNo, InputBufSize, OutputBufSize, BaudRate,
                   Parity, WordLength, StopBits, Protocol),
         openfile(sp,"DATA.TRS"),!,
         process_until_eof, closefile(sp),
         write("\nPrint succeeded"),
         wait_until_empty_output_buffer(100),
         closeRS232(1).

   run :- closeRS232(1).

/**********************************************************************
          Open a file for reading - if possible
**********************************************************************/

   openfile(Sp,FileName) :- openread(Sp,FileName),!, readdevice(Sp).
   openfile(_,FileName) :- write("\nImpossible to open ",FileName),fail.

/**********************************************************************
      Read a character from the current readdevice,
               convert it to uppercase and print it.
         Continue this process until eof or a "time out" event.
**********************************************************************/

   process_until_eof :-
         eof(sp),transmit('\012'),!.                              /* Send Form Feed */
   process_until_eof :-
        readchar(CH), str_char(S,CH), upper_lower(S2,S), str_char(S2,CH2),
        transmit(CH2), process_until_eof.

   transmit('\010') :-
        !,trans_ch_delay('\013',10),trans_ch_delay('\010',10).
   transmit(CH) :- trans_ch_delay(CH,10).

   trans_ch_delay(CH,_) :- txCh_RS232(1,CH),!.
   trans_ch_delay(CH,I) :- I>0,!,I2 = I-1,ticks(50),trans_ch_delay(CH,I2).
   trans_ch_delay(_,_) :-
        closefile(sp), closeRS232(1),
        write("\nPrinter is not ready, program aborted"), fail.

/***********************************************************************
               Wait until output buffer is empty
***********************************************************************/

   wait_until_empty_output_buffer(_) :- queuesize_RS232(1,_,Queue), Queue = 0,!.
   wait_until_empty_output_buffer(I) :-
        I>0,!, I2 = I-1,ticks(10),
        wait_until_empty_output_buffer(I2).
   wait_until_empty_output_buffer(_) :- write("\nTime out").
```

Terminal Emulation

In this section, you'll see how to construct a terminal-emulation program using the tool predicates in the sample file XTERM.PRO. The program allows your PC to act as a terminal when connected either to another PC (via the COM1 ports of each machine) or to a larger mini- or mainframe computer.

Some of the serial communications tool predicates are used to define a predicate *terminal* that allows interrupt-based transmission and reception. Everything received from COM1 is shown in the *Receive* window, and characters typed at the keyboard are transmitted through COM1 and echoed to the *Transmit* window. This process continues until Escape (ASCII code 27) is pressed, and the program terminates.

XTERM.PRO

```
project "xterm"

include "tdoms.pro"
include "comglobs.pro"
include "tpreds.pro"
include "status.pro"
include "menu.pro"

predicates
   run
   terminal
   chk_rdch
   chk_wrch
   rdch_CRLF_RS232(char)
   trans_ch(char)

goal
   run.

clauses
   run :-
         makewindow(2, 42,36," Transmit window ", 0,0,12,80),
         makewindow(3, 63,5," Receive window ", 12,0,12,80),
         PortNo = 1,                                              /* COM1 */
         InputBufSize = 1,                          /* Size of input buffer */
         OutputBufSize = 1,                        /* Size of output buffer */
         BaudRate = 7,                            /* 9600 bits per second */
         Parity = 0,                                           /* No parity */
         WordLength = 3,                                 /* Eight data bits */
         StopBits = 0,                                    /* One stop bits */
         Protocol = 0,                               /* Fully asynchronous */
         openRS232(PortNo, InputBufSize, OutputBufSize, BaudRate,
                   Parity, WordLength, StopBits, Protocol),
         terminal, !,closeRS232(1).

   run :- closeRS232(1).

/**********************************************************************
                       TERMINAL MODE
                  Transmission without time out
**********************************************************************/

   terminal :- chk_rdch, chk_wrch,terminal.

   chk_rdch :- rdch_CRLF_RS232(CH),!,gotowindow(3), write(CH).
   chk_rdch.

   chk_wrch :-
         gotowindow(2), cursor(R,C),cursor(R,C),not(keypressed),!.
   chk_wrch :- readchar(CH),CH<>'\027', wrch_CRLF(CH).

   rdch_CRLF_RS232(CH) :- rxch_RS232(1,CH), CH<>'\013'.

   wrch_CRLF('\013') :- !,nl, trans_ch('\013'), trans_ch('\010').
   wrch_CRLF(CH) :- write(CH), trans_ch(CH).

   trans_ch(CH) :- txCh_RS232(1,CH),!.
   trans_ch(CH) :- trans_ch(CH).
```

Polled Communication with Time Out

Gluing together all the ideas from the sample programs so far, you can easily build a program that allows the polled transmission and reception of data with a time out feature, and that displays a suitable message in case of any communications problems. The Toolbox provides one called XPOLLING.PRO. Polling simply means that the sending device checks whether the receiving device is ready to receive information.

XPOLLING.PRO

```
Project "xpolling"

include "tdoms.pro"
include "tpreds.pro"
include "menu.pro"
include "comglobs.pro"

predicates
   send_str(STRING)
   send_ch_CRLF(CHAR,INTEGER)
   send_ch(CHAR,INTEGER)
   receive_str()
   receive_ch(CHAR,INTEGER)
   wait_ok(INTEGER,INTEGER,INTEGER)
   wr_status(INTEGER)
   check_status(INTEGER,INTEGER,STRING)

goal
   openRS232(1,256,256,7,0,3,0,2),
   send_str("Hello, all our readers\n"),
   closeRS232(1).

clauses
/* Transmit a string */
   send_str("") :- !.
   send_str(S) :- frontchar(S,CH,S2),
         write(CH), send_ch_CRLF(CH,50),
         send_str(S2).

   send_ch_CRLF('\10',I) :- !,send_ch('\13',I), send_ch('\10',I).
   send_ch_CRLF(CH,I) :- send_ch(CH,I).

   send_ch(CH,_) :- wrch_RS232(1,CH),!.
   send_ch(CH,I) :- status_RS232(1,Status), !,
         wait_ok(Status,I,I2), send_ch(CH,I2).

/* Receive a string and copy it to a file */
   receive_str() :-
         receive_ch(CH,50),!,
         write(CH),
         writedevice(FP), writedevice(df), write(CH), writedevice(FP),
        receive_str().
    receive_str().

    receive_ch(CH,_) :- rdch_RS232(1,CH), CH<>'\013', !.
    receive_ch(CH,_) :- rdch_RS232(1,CH), !.
    receive_ch(CH,I) :- status_RS232(1,Status), !,
        wait_ok(Status,I,I2), receive_ch(CH,I2).

/* Time out */
    wait_ok(_,I,I2) :- I > 0, I2 = I-1,ticks(10),!.
    wait_ok(Status,_,50) :- wr_status(Status).

/* De-mask status value */
    wr_status(0) :- !.
    wr_status(Status) :-
        shiftwindow(WD), shiftwindow(1),
        check_status(Status,1, "Input Characters have been lost"),
        check_status(Status,2,  "Parity Error"),
        check_status(Status,4,  "Overrun detected"),
        check_status(Status,8,  "Framing error detected"),
        check_status(Status,16, "Break signal detected"),
        check_status(Status,32, "An XOFF has been received"),
        check_status(Status,64, "An XON has been received"),
        check_status(Status,128,"An XOFF has been transmitted"),
        check_status(Status,256,"An XON has been transmitted"),
        check_status(Status,512,"Input buffer empty when attempt to read"),
        check_status(Status,1024,"Output buffer full when attempt to write"),
        write("\nPress SPACE to continue or ESC to abort"), readchar(Ch),
        shiftwindow(2), shiftwindow(3),
        shiftwindow(WD),CH<>'\27'.

    check_status(Status,BitMask,Mess) :-
        bitand(Status,BitMask,V), V<>0, !, nl, write(Mess).
    check_status(_,_,_).
```

Transmission with a True Subset of the XMODEM Protocol

This section contains a program, XXMODEM.PRO, that transmits and receives data in *packets* according to a true subset of the XMODEM protocol. The packet format is

```
    Packet No, Complement to Packet No, ...128 bytes of data ..., Checksum
```

XMODEM is a file-transfer transmission standard that performs error checking. After a block of data is transferred, a *checksum* (the sum of the ASCII codes of all the characters in the block) is sent. The receiving computer computes a checksum of the block it receives and compares it with the checksum sent. The following transmission abbreviations are used:

ACK	ACKnowledgment
NAK	Negative AcKnowledgment
SOH	Start Of data cHaracter
EOT	End Of Transmission

The sequence of events between transmitter and receiver can be summarized as follows:

TRANSMITTER	**RECEIVER**
	Send NAK
Send SOH	
Send packet	
	Everything is ok : Send ACK
	Everything is not ok : Send NAK

(This sequence is repeated until the transmission is accepted. When all the required packets have been transmitted, the transmitter takes the lead again.)

Send EOT	
	Send ACK

If something goes wrong during transmission (say, a checksum error is detected by the receiver), then the tool predicates *delInBuf* and *delOutBuf* are used to empty the buffers before transmission is re-attempted.

XXMODEM.PRO

```
project "xxmodem"

domains
   FILE = sp;dp

database
   last_char(Char)
   retrans_coun(Integer)

include "tdoms.pro"
include "comglobs.pro"
include "tpreds.pro"
include "menu.pro"

domains
   Package = package(char,char,dataL)
   DataL = char*

predicates
   run
   decide(integer)

   send_file(char)
   receive_file(char)
   reset_last_char
   read_list(integer,dataL)
   mkList(integer, dataL)
   write_list(dataL)

/* Transmission predicates */
   Send_Package(package)
   send_data(integer,dataL,char)
   reset_retransmit_counter
   increment_retransmit_counter
   send_and_wait(char,char,integer)

   receive_package(char,char,package)
   wait_trans_ready
   receive_data(integer,char,dataL)
   rxch_RS232_delay(integer,char)                      /* Receive with a time out */
   check_notEOT

   check_next_char(char)
   ignore_until_received(char)
   test_for_nak

   mess(string,char)
   assert_char(char)
   headline(char)
   opd_headline(integer,char,char)

goal
   run.

clauses
   run :-
         makewindow(3, 81,21," Package window ", 0,44,12,36),
         makewindow(1, 42,36," Message window ", 0,0,12,44),
         makewindow(2, 63,5," Transmission window ", 12,0,12,80),

         PortNo = 1,                                               /* COM1 */
         InputBufSize = 256,                          /* Size of input buffer */
         OutputBufSize = 256,                        /* Size of output buffer */
         BaudRate = 7,                                /* 9600 bits per second */
         Parity = 0,                                             /* No parity */
         WordLength = 3,                                    /* Eight data bits */
         StopBits = 0,                                        /* One stop bits */
         Protocol = 2,                            /* XON/XOFF can not be used */
         openRS232(PortNo, InputBufSize, OutputBufSize, BaudRate,
                   Parity, WordLength, StopBits, Protocol),
         repeat,
         menu(10,20,64,23,
              ["Transmit Data.TRS",
               "Receive Data.RCV",
               "Quit"],
               "Choose an option",Choice),
         decide(Choice),
         fail.
```

```
/* Transmit a file using packaging */
   decide(1) :-
         openread(sp,"DATA.TRS"), readdevice(sp),
         send_file('\001'), !, closefile(sp).
   decide(1) :- closefile(sp).

/* Receive a file using packaging */
   decide(2) :-
         openwrite(dp,"DATA.RCV"), writedevice(dp),
         receive_file('\001'), !, closefile(dp).
   decide(2) :- closefile(dp).

/* Quit */
   decide(3) :- closeRS232(1), exit.

/***********************************************************************
                          Transmit a file
***********************************************************************/

   send_file(Pno) :-
         read_list(0,DataL),!, char_int(PNO,V), V2 = -V, char_int(CPNO,V2),
         send_package(package(PNO,CPNO,DataL)),
         V3 = V+1, char_int(PNO2,V3),
         send_file(PNO2).

   send_file(_) :-

/* read_list (read characters from a file) failed */
/* Everything is ok--send an EOT and wait for ACK */
         mess("Send EOT and wait on ACK",' '),
         send_and_wait('\04','\06',5).

/***********************************************************************
                           Receive a file
***********************************************************************/

   receive_file(PNO) :-
         char_int(PNO,V), V2 = -V, char_int(CPNO,V2),
         reset_last_char,
         receive_package(PNO,CPNO,package(_,_,DATAL)),!,
         write_list(DATAL),
         V3 = V+1, char_int(PNO2,V3),
         receive_file(PNO2).

   receive_file(_).

/* received_package failed, that means a EOT has been received */
/* or the transmission is in time out */
   reset_last_char :- not(last_char(_)),!.
   reset_last_char :- retract(last_char(_)),!.

/***************************************************************************
      Support predicates for send_file and receive_file

read_list: If possible, read 128 bytes from file and convert it to a list.
write_list: Write a list of characters to the current output device
***************************************************************************/
```

```
   read_list(128,[]) :- !.
   read_list(I,[H|T]) :- readchar(H), !, I2 = I+1, read_list(I2,T).
   read_list(I,L) :- I>0, Len = 128-I, mkList(Len,L).

   mkList(0,[]) :- !.
   mkList(I,['\026'|T]) :- I2 = I-1, mkList(I2,T).

   write_list([]) :- !.
   write_list([H|T]) :- write(H), write_list(T).

/**********************************************************************
                          Transmit a package
**********************************************************************/

   send_Package(package(PNO,CPNO,DataL)) :- headline(PNO),

/* Wait for a NAK from the receiver */
         mess("Wait for NAK ...",' '),
         ignore_until_received('\021'), delInbuf_RS232(1),

/* Send SOH */
         txCh_RS232(1,'\001'),
         mess("Send SOH",' '), ticks(20), delInbuf_RS232(1),

/* Send package number and the complements */
         mess("Transmit PNO ",PNO),
         txCh_RS232(1,Pno), ticks(1), test_for_nak,
         mess("Transmit CPNO ",CPNO),
         txCh_RS232(1,CPNO), ticks(1), test_for_nak,

         mess("Transmit Data ...",' '),
         send_data(0,DataL,'\0'),                    /* Send data and checksum */
         mess("Wait for ACK ...",' '),
         check_next_char('\006'),                    /* Wait for acknowledgment */
         reset_retransmit_counter,!.

   send_Package(package(PNO,CPNO,dataL)) :-

/* Something has gone wrong */
/* We will retransmit the same package five times at most */
         increment_retransmit_counter,
         mess("Retransmitting package ",PNO),
         delInBuf_RS232(1), delOutBuf_RS232(1),
         send_Package(package(PNO,CPNO,DataL)).

/**********************************************************************
              Retransmission predicates used by send_Package
**********************************************************************/

   reset_retransmit_counter :- not(retrans_coun(_)), !.
   reset_retransmit_counter :- retract(retrans_coun(_)), !.

   increment_retransmit_counter :-
         not(retrans_coun(_)),assertz(retrans_coun(1)),!.
   increment_retransmit_counter :-
         retract(retrans_coun(I)), I<5, I2 = I+1, !,
              assertz(retrans_coun(I2)).
   increment_retransmit_counter :-
         delInBuf_RS232(1), delOutBuf_RS232(1),
        mess("\nError transmitting package, Transmission ABORTED\n",' '),
        fail.

   send_and_wait(CH1,CH2,_) :-
        txCh_RS232(1,CH1),rxch_RS232_delay(10,CH),CH = CH2,!.
   send_and_wait(_,CH2,_) :-
        delInbuf_RS232(1),rxch_RS232_delay(10,CH),CH = CH2.

/***********************************************************************
                    Receive a package (128 characters)
***********************************************************************/

   receive_package(PNO,CPNO,package) :-
        headline(PNO),
        mess("Continue send NAK's and wait for SOH ...",' '),
        wait_trans_ready,                        /* Send NAK until a SOH is received */
        mess("Wait for package number PNO=",PNO),
        check_next_char(PNO),                    /* Check for correct package number */
        mess("Wait for complement number CPNO=",CPNO),
        check_next_char(CPNO),                                 /* And its complement */

        mess("Receive DATA ...",' '),
        receive_data(0,'\0',DataL),!,                  /* Receive data and checksum */

/* Everything is all right - Send an acknowledgment */
        mess("Data ok - send ACK",' '),
        txCh_RS232(1,'\006'),
        Package = package(PNO,CPNO,dataL).

   receive_package(PNO,CPNO,package) :-

/* If wait_trans_ready failed, then check if EOT received */
        mess("Transmission error or reception of EOT",' '),
        check_notEOT,

/* Transmission of current package crashed - Send a NAK */
        delInBuf_RS232(1), delOutBuf_RS232(1),
        mess("It was a transmission error while receiving package PNO=",PNO),
        mess("Send NAK because of error in transmission",' '),
        txCh_RS232(1,'\021'),ticks(10), delInBuf_RS232(1),
        receive_package(PNO,CPNO,package).                            /* Try again */

   check_notEOT :- not(last_char(_)),!.                   /* last_char is updated
                                                              by wait_trans_ready */
   check_notEOT :- last_char(CH),CH<>'\004',retract(last_char(CH)),!.

   assert_char(CH) :- retract(last_char(_)),assertz(last_char(CH)),!.
   assert_char(CH) :- assertz(last_char(CH)),!.

   wait_trans_ready :-

/* Send NAK until reception of SOH */
        txCh_RS232(1,'\021'), ticks(10),

/* If a character is received, it should be SOH */
        rxch_RS232(1,CH), assert_char(CH),
        CH = '\001',!.
   wait_trans_ready :- check_notEOT,!, wait_trans_ready.
   wait_trans_ready :-
```

```
        mess("EOT received - Send ACK",' '),
        txCh_RS232(1,'\006'),fail.                          /*Send ACK after receive EOT*/

/*********************************************************************
           Receive a data block (128 characters).
Fails if a character is not received in the specified time out period.
  Data and the corresponding checksum will be echoed to the screen.
*********************************************************************/

   receive_data(128,CheckSum,[]) :- !,check_next_char(CheckSum).
   receive_data(I,Csum1,[CH|T]) :- rxch_RS232_delay(50,CH),

/* Compute the checksum on the fly */
        char_int(Csum1,V1), char_int(Ch,V2),
        V3 = V1+V2, char_int(Csum2,V3),
        I2 = I+1, opd_headline(I2,Csum2,CH),
        receive_data(I2,Csum2,T).

/* Receive characters from COM1 with a 5-second time out period */
   rxch_RS232_delay(_,CH) :- rxch_RS232(1,CH),!.
   rxch_RS232_delay(I,CH) :- I>0,!,I2 = I-1,ticks(5),rxch_RS232_delay(I2,CH).

/*********************************************************************
                        Transmit a data block.
                        Fails if the receiver sends a NAK.
                        Data will be echoed to the screen.
*********************************************************************/

   send_data(_,[],CheckSum) :- !,txCh_RS232(1,CheckSum).
   send_data(I,[H|T],Csum1) :-
        test_for_nak,
        txCh_RS232(1,H),

        char_int(H,V), char_int(Csum1,V1),
        V2 = V+V1, char_int(Csum2,V2),
        I2 = I+1, opd_headline(I2,Csum2,H),
        send_data(I2,T,Csum2).

   test_for_nak :-
        rxch_RS232(1,CH),CH = '\021',!,mess("Received NAK",''),fail.
   test_for_nak.

/*********************************************************************
                         Miscellaneous predicates
*********************************************************************/

   ignore_until_received(CH) :- check_next_char(CH),!.
   ignore_until_received(CH) :- ignore_until_received(CH).

   check_next_char(CH) :- rxch_RS232(1,CH1),!,CH1 = CH.
   check_next_char(CH) :- check_next_char(CH).

   mess(S,CH) :-
        writedevice(WD), writedevice(screen),
        shiftwindow(W), shiftwindow(1),
        char_int(CH,V), write("\n",S,V),
        shiftwindow(W),
        writedevice(WD),
        readdevice(ID),readdevice(keyboard),readdevice(ID).
```

```
headline(PNO) :-
      writedevice(WD), writedevice(screen),
      gotowindow(3), cursor(R,_), cursor(R,33),
      char_int(Pno,V), write("\nPackage NO:",V," Data:"),
      writedevice(WD).

opd_headline(I,Checksum,CH) :-
      writedevice(WD), writedevice(screen),
      gotowindow(2), write(CH),
      gotowindow(3), char_int(Checksum,V),
      cursor(R,C), write(I," Checksum:",V), cursor(R,C),
      writedevice(WD).
```

Modem Communication

This section explains how to use the modem-related tools to send a break signal, set the modem mode, send commands, and send and receive information.

Sending a Break Signal to a Modem

The break signal is used in serial communications to either "wake up" the remote device or terminate a dialog. The tool predicate *sendBreak_RS232* sends a break signal to the I/O port indicated by its single integer parameter (1 means COM1, 2 means COM2). *sendBreak_RS232* fails if the specified port has not been opened. It is declared as follows:

```
Determ sendBreak_RS232 - language c
```

The receiving PC detects a break signal using the *status_RS232* tool predicate.

Setting the Modem Mode

Communication with a Hayes or Hayes-compatible modem involves two basic modes of operation. There are the command strings sent from the computer to the modem to establish a connection (sending a break signal, dialing the telephone number of the remote computer, and the like). Then there is the transmission of *pure* data, such as a file-transfer operation. The following section deals with the predicates that perform both operations. But first, you must set the modem mode.

The tool predicate *setModemMode* sets the communication mode for a Hayes-compatible modem. Before calling *setModemMode,* COM1 or COM2 must already have been initialized using the *open_RS232* tool predicate. Indeed, *setModemMode* fails if the specified port has not been opened. It must be declared in any containing program by a declaration of the form

```
Determ setModemMode(Integer, String, Char, Integer)
                 - (i,i,i,i) language c
```

so that a call of the form

```
setModemMode(PortNo,CommandAtt,CommandTerminator,BreakTime)
```

sets the modem mode as follows:

PortNo	= 1 means COM1 serial communication port.
	= 2 means COM2 serial communication port.
CommandAtt	= Modem command prefix—normally AT, which means the modem should expect a command.
	= Null string (that is, "") means the modem should expect data.
CommandTerminator	= Command suffix—normally *CR ('\13').*
	= '\0' denotes no data terminator.
BreakTime	= A number in the range 0 to 32767 denoting the length of time (in hundredths of a second) for which a break signal is to be placed on the line; normally in the range 10 to 25.

The predicate *set_modem* (defined as follows) can be used to switch between modes in a menu-driven program for a modem connected to COM2:

```
/* Send a break signal */
   set_modem("break") :- !,setModemMode(2,"AT",'\013',10),sendBreak_RS232(2).

/* Prefix every command with "AT" and suffix it with CR */
   set_modem("at on") :- !,setModemMode(2,"AT",'\013',10).

/* No transformation at all - Useful when transmitting data */
   set_modem("at off") :- !,setModemMode(2,"",'\000',10).
   set_modem(_) :- wrstr_modem(L,_).
```

Sending a Command or Data to a Modem

The tool predicate *txStr_Modem* sends a command or pure data string to the modem using the parameters set by the most recent call to the *setModemMode* predicate—in particular, the string is sent via the serial I/O port affected by that call. *txStr_Modem* fails if the modem port is not initialized. It should be declared in any containing program as follows:

```
Determ txStr_Modem(String,Integer) - (i,o) language c
```

A call of the form

```
txStr_Modem(TxString,NoOfCharsTransmitted)
```

with *TxString* bound to the command or data string to be transmitted binds *NoOfCharsTransmitted* to the actual number of characters transmitted. This may be different from the number of characters in *TxString*, since a telephone line can be faulty—for instance, because of noise on the telephone line. sometimes, you may have to retransmit some or all of *TxString*. Also, in a modem command string, extra characters are added to the string according to the settings of the parameters in the most recent call to *setModemMode.*

Note that the length of the string transmitted can never be larger than the size of the output buffer.

For example, the following clause defines the predicate *dial,* which calls the telephone number of a remote computer:

```
dial(No) :-
      txStr_Modem("Z ",_),                         /* Reset the modem to initial state*/
      ticks(100),                                  /* Wait for command to be executed */
      txStr_Modem("C1",_),                                             /* Carrier on */
      ticks(100),
      txStr_Modem("H1",_),                                              /* Hang up */
      ticks(100),
      concat("D ",No,DialNo),
      txStr_Modem(DialNo,_),                         /* The remote modem must be in
                                                   auto-answer mode for this to work */
      ticks(2000).                                 /* There should now be a connection */
```

Receiving a Response from a Modem

The tool predicate *rxStr_Modem* takes a single string parameter that is bound to the characters received (if any) from the remote modem. If there is a command terminator (see *setModemMode*) in the input buffer or any preceding *AT*tention character, the returned string won't include these extra characters—unless the command terminator is the NULL character ('\000'), in which case everything in the input buffer is returned in the string parameter. *rxStr_Modem* fails if a serial I/O port has not been opened for the modem.

In any containing program, *rxStr_Modem* should be declared as follows:

```
   rxStr_Modem(String) - (o) language c
```

The following example shows how *txStr_Modem* and *rxStr_Modem* can be used to obtain an error code:

```
   get_errorcode :-
         txStr_Modem("I1",_),                              /* Ask for error code */
         ticks(100),
         rxStr_Modem(Response1),                            /* Should be 'I1' */
         ticks(100),
         rxStr_Modem(Response2),                       /* Should be error code */
         ticks(100),
         rxStr_Modem(Response3).                            /* Should be 'OK' */
```

A Menu-Driven Serial Communications Program

In this section, all the serial communications tool predicates are combined into a complete serial communications package (XCOMMU.PRO). This package has a very smart menu-driven user interface, constructed with tools from chapters 2 and 3. The Toolbox's serial communications tool predicates make it surprisingly easy to build.

XCOMMU.PRO

```
/*************************************************************************
                    Complete serial communications package
*************************************************************************/

nobreak

project "xcommu"

domains
   FILE  = sf; df
```

```
database
   editbuf(string)
   port(integer,string)

include "tdoms.pro"
include "comglobs.pro"
include "tpreds.pro"
include "menu.pro"

predicates
   decide(integer)

/* Polled transmission with time out */
   send_str(string,integer)
   send_ch(char,integer,integer)
   send_ch_CRLF(char,integer,integer)
   receive_str(integer)
   receive_ch(char,integer,integer)
   wait_ok(integer,integer,integer)

/* Interrupt-based Terminal Emulation */
   interactive_com
   chk_rdch
   chk_wrch
   rdch_CRLF_RS232(integer,char)

/* Interrupt-based Terminal Modem Communication */
   interactive_modem
   chk_rdmodem
   chk_wrmodem
   chk_modem(string,string)
   init_modem_line
   chk_modem_delay(integer,integer)
   send_str_modem(string,integer,integer)
   trans_modem(string,integer,integer,integer)
   receive_modem(integer,integer,integer)

/* Read and write to console the status of transmission */
   wr_status(integer)
   chk_stat(integer,integer,string)

/* Miscellaneous */
   mess(string)
   rdch_keyb(char)
   get_FileName(string,string)

goal
   makewindow(1, 23,130," Message window ", 4,35,8,45),
   makewindow(2, 42,36," Transmit window ", 3,0,10,80),
   makewindow(3, 63,5," Receive window ", 13,0,10,80),
   makewindow(6, 10,7," Configuration ",0,0,3,80),
   assertz(editbuf("")),
   asserta(port(1,"COMMU")),
   asserta(port(2,"MODEM")),
   repeat,
   port(ComPort,"COMMU"),
   port(ModPort,"MODEM"),
   shiftwindow(6), clearwindow,
   write(" The communication port is COM",ComPort,",  The modem is COM",ModPort),
   shiftwindow(2), shiftwindow(3),
   menu(10,20,64,23,
        ["Open communication port",                                          /* 1 */
         "Close communication port",                                         /* 2 */
         "Send File using Protocol",                                         /* 3 */
         "Receive File using Protocol",                                      /* 4 */
         "Terminal Mode",                                                    /* 5 */
         "No of Characters in buffers",                                      /* 6 */
         "",
         "Initialize modem port",                                            /* 8 */
         "Close modem port",                                                 /* 9 */
         "Send File using modem",                                           /* 10 */
         "Receive File using modem",                                        /* 11 */
         "Terminal Mode using modem",                                       /* 12 */
         "",
         "Editor",                                                          /* 14 */
         "Operating system",                                                /* 15 */
         "Switch COM PORTS",                                                /* 16 */
         "Quit"],                                                           /* 17 */
         "Choose an option",0,Choice),
   decide(Choice),
   fail.

clauses
/* Open communication port */
   decide(1):-
         port(PortNo,"COMMU"),                                /* COM Port is PortNo */
         InputBufSize = 256,                                /* Size of input buffer */
         OutputBufSize = 256,                              /* Size of output buffer */
         BaudRate = 7,                                     /* 9600 bits per second */
         Parity = 0,                                                  /* No parity */
         WordLength = 3,                                         /* Eight data bits */
         StopBits = 0,                                             /* One stop bits */
         menu(10,10,64,23,["Without RTS/CTS and XON/XOFF",
                           "XON/XOFF without RTS/CTS",
                           RTS/CTS without XON/XOFF",
                           "RTS/CTS and XON/XOFF"],
                           "Choose an option",0,Choice),
        Protocol = CHOICE-1,
        openRS232(PortNo,InputBufSize,OutputBufSize,BaudRate,Parity,
                  WordLength,StopBits,Protocol),!.

   decide(1) :-
         mess("Open RS232 failed").

/* Close communication port */
   decide(2) :-
         port(PortNo,"COMMU"),closeRS232(PortNo),!.                 /* Close PortNo */
   decide(2) :-
         mess("Close RS232 failed").

/* Send file using protocol */
   decide(3) :-
         port(PortNo,"COMMU"),
         shiftwindow(2),
         get_filename("Name of file to be transmitted: ",FileName),
         file_str(FileName,S), send_str(S,PortNo),!.
   decide(3) :- mess("Transmission failed").
```

```
/* Receive file using protocol */
   decide(4) :-
         port(PortNo,"COMMU"),
         shiftwindow(3),
         get_filename("Name of file to be received: ",FileName),
         openwrite(df,FileName), receive_str(PortNo), closefile(df),!.
   decide(4) :- mess("Transmission failed").

/* Terminal Mode */
   decide(5) :-
         shiftwindow(2), write("\nTerminal Mode, Press Esc to abort\n"),
         interactive_com.

/* Number of characters in buffers */
   decide(6) :-
         port(PortNo,"COMMU"),
         queuesize_RS232(PortNo,CharInput,CharOutput),!,
         makewindow(5,109,82," Information ",6,20,5,50),
         write("\nNo of Characters in input buffer  : ",CharInput),
         write("\nNo of Characters in output buffer : ",Charoutput),
         readchar(_), removewindow.
   decide(6) :- mess("No Queues").

/* Initialize modem port */
   decide(8) :-
         port(PortNo,"MODEM"),                         /* PortNo is modem port */
         InputBufSize = 256,                           /* Size of input buffer */
         OutputBufSize = 256,                          /* Size of output buffer */
         BaudRate = 4,                                 /* 1200 bits per second */
         Parity = 2,                                   /* Even parity */
         WordLength = 2,                               /* Seven data bits */
         StopBits = 0,                                 /* One stop bits */
         Protocol = 3,                                 /* RTS/CTS and XON/XOFF */
         openRs232(PortNo,InputBufSize,OutputBufSize,BaudRate,Parity,
                  WordLength,StopBits,Protocol),
         SetModemMode(PortNo,"AT",'\013',25),!.

   decide(8) :- mess("Initialization of MODEM port failed").

/* Close modem port */
   decide(9) :-
          port(PortNo,"MODEM"),
          closeRS232(PortNo),!.                        /* Close PortNo */
   decide(9) :- mess("Close Modem Port failed").

/* Send file using modem port */
   decide(10) :-
         port(PortNo,"MODEM"),
         shiftwindow(2),
         get_filename("Name of file to be transmitted: ",FileName),
         file_str(FileName,S),
         send_str_modem(S,PortNo,1),!.
   decide(10) :- mess("Transmission via modem failed").

/* Receive file using modem port */
   decide(11) :-
         port(PortNo,"MODEM"),
         shiftwindow(3),
         get_filename("Name of file to be received: ",FileName),
        openwrite(df,FileName),
        receive_str(PortNo), closefile(df), !.
    decide(11) :- mess("Transmission via Modem failed").

/* Set modem in Terminal Mode */
    decide(12) :-
        shiftwindow(2),
        init_modem_line,
        write("\nTerminal Mode, Press Esc to abort\n"),
        interactive_modem.

/* Editor */
    decide(14) :-
        makewindow(5,109,82," Edit Window ",6,10,15,60),
        editbuf(Str), edit(Str,Str2),
        retract(editbuf(Str)), assertz(editbuf(Str2)),!.

/* Operating system */
    decide(15) :- system("").

/* Option to switch comm ports commu = 2 and modem = 1 */
    decide(16) :-
       retract(port(_,_)), fail.
    decide(16) :-
       asserta(port(1,"MODEM")),
       asserta(port(2,"COMMU")).

/* Quit */
    decide(17) :-
         closeRS232(1),fail.                              /* Close COM1 */
    decide(17) :-
         closeRS232(2),fail.                              /* Close COM2 */
    decide(17) :- exit.


/**********************************************************************
                  Polled transmission with time out
**********************************************************************/

/* Transmit a string */
    send_str("",_) :- !.
    send_str(S,PortNo) :-
        frontchar(S,CH,S2),
        write(CH), send_ch_CRLF(CH,50,PortNo),
        send_str(S2,PortNo).

    send_ch_CRLF('\10',I,PortNo) :-
        !,send_ch('\13',I,PortNo), send_ch('\10',I,PortNo).
    send_ch_CRLF(CH,I,PortNo) :- send_ch(CH,I,PortNo).

    send_ch(CH,_,PortNo) :- txCh_RS232(PortNo,CH),!.
    send_ch(CH,I,PortNo) :-
        status_RS232(PortNo,Status), !,
        wait_ok(Status,I,I2), send_ch(CH,I2,PortNo).

/* Receive a string and copy it to a file */
    receive_str(PortNo) :-
        receive_ch(CH,50,PortNo),!, write(CH),
        writedevice(FP), writedevice(df), write(CH), writedevice(FP),
        receive_str(PortNo).
   receive_str(_).

   receive_ch(CH,_,PortNo) :- rxch_RS232(PortNo,CH), CH<>'\013', !.
   receive_ch(CH,_,PortNo) :- rxch_RS232(PortNo,CH), !.
   receive_ch(CH,I,PortNo) :-
        status_RS232(PortNo,Status),
        wait_ok(Status,I,I2), receive_ch(CH,I2,PortNo).
```

/* Test for time out */

```
   wait_ok(_,I,I2) :- I > 0, I2 = I-1,ticks(10),!.
   wait_ok(Status,_,50) :- wr_status(Status).
```

/* Transmit a string using modem port */

```
   send_str_modem("",_,_) :- !.
   send_str_modem(S,PortNo,ChCoun) :-
        frontchar(S,CH,S2),
        write(CH), send_ch_CRLF(CH,50,PortNo),
        chk_modem_delay(ChCoun,NewChCoun),
        send_str_modem(S2,PortNo,NewChCoun).
```

/* Some modem are without handshake */

```
   chk_modem_delay(10,1) :- !,Ticks(8).
   chk_modem_delay(I,I2) :- I2 = I+1.
```

/* De-mask status value */

```
   wr_status(0) :- !.
   wr_status(Status) :-
        shiftwindow(WD), shiftwindow(1),
        chk_stat(Status,1,  "Input Characters have been lost"),
        chk_stat(Status,2,  "Parity Error"),
        chk_stat(Status,4,  "Overrun detected"),
        chk_stat(Status,8,  "Framing error detected"),
        chk_stat(Status,16, "Break signal detected"),
        chk_stat(Status,32, "An XOFF has been received"),
        chk_stat(Status,64, "An XON has been received"),
        chk_stat(Status,128,"An XOFF has been transmitted"),
        chk_stat(Status,256,"An XON has been transmitted"),
        chk_stat(Status,512,"Input buffer empty when attempt to read"),
        chk_stat(Status,1024,"Output buffer full when attempt to write"),
        write("\nPress Space to continue or Esc to abort"), readchar(Ch),
        shiftwindow(2), shiftwindow(3),
        shiftwindow(WD),CH<>'\27'.

   chk_stat(Status,BitMask,Mess) :-
        bitand(Status,BitMask,V), V<>0, !, nl, write(Mess).
   chk_stat(_,_,_).
```

/**

TERMINAL MODE

Interrupt-based transmission without time out

**/

/* Terminal Mode */

```
   interactive_com :- chk_rdch, chk_wrch,interactive_com.

   chk_rdch :-
       port(PortNo,"COMMU"),
        rdch_CRLF_RS232(PortNo,CH),!,shiftwindow(3), write(CH).
    chk_rdch.

    chk_wrch :- shiftwindow(2), cursor(R,C), cursor(R,C), not(keypressed),!.
    chk_wrch :-
          port(PortNo,"COMMU"),!,
          rdch_keyb(CH),CH<>'\027',
          write(CH),
          txCh_RS232(PortNo,CH).

    rdch_CRLF_RS232(PortNo,CH) :- rxch_RS232(PortNo,CH), CH<>'\013',!.
    rdch_CRLF_RS232(PortNo,CH) :- rxch_RS232(PortNo,CH).


/*********************************************************************
                TERMINAL MODE - MODEM COMMUNICATION
                Interrupt-based transmission without time out
*********************************************************************/

    interactive_modem() :- chk_rdmodem, chk_wrmodem,interactive_modem.

    chk_rdmodem :- RxStr_modem(Mess),shiftwindow(3), write(Mess), fail.
    chk_rdmodem.

    chk_wrmodem :- shiftwindow(2), cursor(R,C), cursor(R,C), not(keypressed),!.
    chk_wrmodem :- readln(L), upper_lower(L,L2), chk_modem(L,L2).

/* Command to the modem when it is in terminal mode */
/* Send a break signal */
    chk_modem(_,"break") :- !,
            port(PortNo,"MODEM"),!,
            SetModemMode(PortNo,"AT",'\013',10),SendBreak_RS232.

/* Prefix every commands with "AT" and suffix it with CR */
    chk_modem(_,"at on") :- !,
            port(PortNo,"MODEM"),!,
            SetModemMode(PortNo,"AT",'\013',10).

/* No transformation at all - might be useful when transmitting data */
    chk_modem(_,"at off") :- !,
            port(PortNo,"MODEM"),!,
            SetModemMode(PortNo,"",'\013',10).                /* No transform at all */
    chk_modem(L,_) :- TxStr_modem(L,_).

/*********************************************************************
                      MODEM SUPPORT PREDICATES
*********************************************************************/

    init_modem_line :-
          shiftwindow(OldWD),                                        /* Old window */
          trans_modem("Z ",2,1,10),               /* Reset the modem to initial state */
          trans_modem("C1",2,1,10),                             /* Set carrier high */
          shiftwindow(OldWd).

    trans_modem(Mstr,NoofAnsw,NoOFRetr,Delay) :-
          ticks(10),
          shiftwindow(2),                                  /* Transmission window */
          write("\n" ,Mstr),
          TxStr_modem(Mstr,_),                                 /* Command to modem */
        shiftwindow(3),                                          /* Receive window */
        receive_modem(NoofAnsw,NoOfRetr,Delay).

   receive_modem(0,_,_) :- !.                                    /* No more to receive */
   receive_modem(I,R,Delay) :-
        ticks(Delay),
        RxStr_modem(Mess1),
        write(Mess1), I2=I-1, !,
        receive_modem(I2,R,50).                                  /* First delay is highest */

   receive_modem(_,1,_) :- trans_modem("I2",5,0,400),!,fail.

/***********************************************************************
                                Miscellaneous
***********************************************************************/

   mess(Str) :-
        shiftwindow(WD), shiftwindow(1),
        write("\n\n",Str),
        write("\nPress Space to continue"), readchar(_),
        shiftwindow(2), shiftwindow(3),
        shiftwindow(WD).

/* Read char from keyboard and transform CR to LF */
   rdch_keyb(CH) :- readchar(CH), CH<>'\013',!.
   rdch_keyb('\010').                                            /*CH = '\010'.*/

/* Get file name from console */
   get_FileName(Mess,FileName) :-
        makewindow(4,12,52," Input ",12,10,3,50),
        write(Mess),
        readln(FileName),FileName<>"",!,removewindow.
   get_FileName(_,FileName) :-
        makewindow(4,23,12," Input ",12,10,12,50),
        disk(Disk), dir(Disk,"*.txt",FileName),!,removewindow,removewindow.
   get_FileName(_,"") :-
        removewindow, removewindow, fail.
```

C H A P T E R

6

Importing Data from Other Systems

With the tool predicates described in this chapter, it is possible to read databases generated by the well-known database managers Reflex and dBASE III, and to read specific cell values from the spreadsheets Lotus 1-2-3 and Symphony. This means you can import records generated by these packages and manipulate them in your Turbo Prolog application program.

To access a Reflex file from a Turbo Prolog program, you need to **include** the file REFLEX.PRO. Similarly, to access a dBASE III file, **include** the file DBASE3.PRO; to access a 1-2-3 or Symphony file, **include** the file LOTUS.PRO. (Remember modular programming is explained in Appendix A, "Compiling a Project.")

In all three cases, it is also necessary to link in the module REALINTS.OBJ provided on the distribution disk. REALINTS.OBJ performs a conversion between four integers and a real; the source for REALINTS.OBJ is defined in REALINTS.C. You should have Turbo Prolog version 1.10 or later to use these tool predicates, because they utilize binary file access.

All tools in this section require READEXT.PRO, which performs general conversions.

Let's begin with importing Reflex files, move on to dBASE III files, then finish with Lotus 1-2-3 and Symphony files.

Accessing a Reflex File

To access a Reflex file, you must first call the tool predicate *Init_Reflex,* which builds Prolog data structures describing the Reflex records. It's found in the file REFLEX.PRO, and its declaration takes the form

```
Init_Reflex(INTEGER,FLDNAMES,REFLEXTYPEL,TEXTPOOLS)
```

The corresponding nonstandard domain declarations are as follows:

```
FLDNAMES = STRING                                   /* Reflex field names */

REFLEXTYPEL = REFLEXTYPE*
REFLEXTYPE  = u; t; rt; d; r; i                     /* Internal Reflex type
                                                        for each field */

TXTPOOLS = TXTPOOL*
TXTPOOL = REPTXT*
REPTEXT = text(INTEGER,STRING)                      /* Indexed strings */
```

Using the tool predicate *Rd_ReflexFile,* all the Reflex data records in a given file can be read and collected into a single Turbo Prolog list all at once. This list belongs to the tool domain *REFLEXRECL,* which is declared as follows:

```
REFLEXRECL = REFLEXREC*                  /* The database is a list of records */
REFLEXREC = REFLEXELEM*                    /* A record is a list of elements */
REFLEXELEM = date(INTEGER);                /* 16-bit int. representing number */
                                         /* of days since December 31, 1899 */
             real(REAL);                   /* 64-bit IEEE floating-point real */
             int(INTEGER);                            /* 16-bit signed integer */
             text(STRING);                         /* A string representing a text */
             untyped;                                     /* No data stored */
             error
```

Together, a record list from the domain *REFLEXRECL* and the list of field names from the domain *FLDNAMES* form a complete data structure describing a Reflex database.

The sample Reflex file, XREFLEX.RXD, is a personnel database constructed so that Reflex displays its structure as shown in the following Reflex Field & Sort Settings tool:

Field	Type	Formula	Sort #	A/D	Format	Prec
Name	Text		2	a		
Birth Date	Date				dd-mmm-	
Department	Repeating Text					
Salary	Numeric		1	D	Currenc	2
Age	Integer	22			Fixed	0
Marital Status	Repeating Text					

A corresponding Turbo Prolog record from the domain *REFLEXREC* has the form:

```
[text("Frank Borland"),                              /* Name field */
 date(4722),                      /* No. of days since Dec. 31, 1899 */
 text("Mascot"),                                     /* Department */
 real(102),                                              /* Salary */
 int(73),                                                   /* Age */
 text("None")]                                   /* Marital status */
```

(NOTE: Each record is a list inside the list of records.)

The corresponding list of field names from the tool domain *FLDNAMES* is:

```
["Name", "Birth Date", "Department",
 "Salary", "Age", "Marital Status"]
```

The declaration of *Rd_ReflexFile* is as follows:

```
Rd_ReflexFile(INTEGER,REFLEXTYPEL,TEXTPOOLS,REFLEXRECL)
```

with all parameters except the last being input parameters, obtained from the call to *Init_Reflex*. The following example shows how to access the

Reflex file XREFLEX.RXD from a Turbo Prolog program, using *init_Reflex* and *rd_ReflexFile*:

```
   AccessAll(ReflexRecs) :-
         openread(fp,"XREFLEX.RXD"), readdevice(fp),
         filemode(fp,0),                                       /* Binary mode */

/* Build data structure */
         init_Reflex(TotRecs,FldNames,TypeL,TextPools),

/* Read all data records */
         rd_DataRecs(TotRecs,TypeL,TextPools,ReflexRecs),

/* List the records */
         makewindow(85,41,36," Reflex(tm) All Data Records ",
                    0,0,25,40),
         list_Recs(FldNames,ReflexRecs), DoPrompt.
```

The two last literals (*makewindow* and *list_Recs*) are not necessary, but they display the contents of the Reflex file in a readable manner.

Reading One Reflex Record at a Time

If the Reflex file to be read is very large, you may find that it takes too much time to read all the records or the resulting list is too big to load into memory. Fortunately, a tool predicate is provided that makes it possible to read one record at a time, do some computation, remove the storage allocation via backtracking if desired, and then read the next record.

In order to access Reflex records singly, it is first necessary to call the tool predicate *init_Reflex*, which calls the first record. The tool predicate that allows access to the next Reflex record is *Rd_ReflexRec*, which has this declaration:

```
   Rd_ReflexRec(INTEGER,REFLEXTYPEL,TEXTPOOLS,REFLEXREC)
```

The following example shows you how to use *Init_Reflex* and *Rd_ReflexRec* to read a Reflex file record by record:

```
   OneByOne :-
         openread(fp,"Xreflex.rxd"), readdevice(fp),
         filemode(fp,0),

/* Build data structure */
         Init_Reflex(TotRecs,FldNames,TypeL,TextPools),
         makewindow(85,72,33," Reflex(tm) Sequential Access ",
                    0,40,25,40),
/* Read one by one */
        Rd_ReflexRec(TotRecs,TypeL,TextPools,Rec),

/* Do some computations using the record Rec ... */
/* List the record */
        nl,nl,list_rec(FldNames,Rec),
        PressAKey, fail.
```

The predicate *Rd_ReflexRec* is non-deterministic: While there are still records in the Reflex file, it reads the next record in the structure *Rec* and then returns from the call. When the calling predicate fails, *Rd_ReflexRec* generates a new solution by reading the next Reflex record.

XREFLEX.PRO contains the program shown below, which is an expansion of the previous examples into a complete program.

XREFLEX.PRO

```
project "xreflex"

domains
   FILETAB = fp

global predicates
   real_ints(REAL,INTEGER,INTEGER,INTEGER,INTEGER) - (o,i,i,i,i) language c

include "readext.pro"
include "reflex.pro"

predicates
/* List data records */
   list_Recs(FldNames,ReflexRecL)
   list_rec(FldNames,ReflexRec)
   list_elem(ReflexElem)
   PressAKey
   doPrompt

clauses
   list_Recs(_,[]) :- !.
   list_Recs(FldNames,[ReflexRec|ReflexRecs]) :-
        nl,nl,
        list_rec(FldNames,ReflexRec),
        PressAKey,
        list_Recs(FldNames,ReflexRecs).

   list_rec([],[]) :- !.
   list_rec([FldName|FldNames],[Elem|Elems]) :-
        writef("\n%-20: ",FldName),
        list_elem(Elem),
        list_rec(FldNames,Elems).

   list_elem(untyped) :- write("Untyped").
   list_elem(text(Str)) :- write(Str).
   list_elem(date(Date)) :- write(Date).
   list_elem(real(Real)) :- write(Real).
   list_elem(int(Int)) :- write(Int).

   PressAKey :-
         makewindow(_,_,_,_,MinR,_,NoofR,_),trace(on),
         cursor(R,_), R< = MinR+NoofR-4,!.

   PressAKey :- doPrompt,cursor(R,C), scroll(R,0),cursor(0,C).

   doPrompt :-
         makewindow(Nr,Att,_,_,MinR,MinC,NoofR,NoofC),
         MinR2 = MinR+NoofR-1, MinC2 = MinC+NoofC/3+1,
         str_len(" Press a key",Len), Len2 = Len+1, bitxor(Att,8,Att2),
         makewindow(Nr,Att2,0,"",MinR2,MinC2,1,Len2),
         write(" Press a key"),
         readdevice(FP), readdevice(keyboard), readchar(_), readdevice(FP),
         removewindow.

/*********************************************************************
                                Goal
*********************************************************************/
goal
   openread(fp,"xreflex.rxd"), readdevice(fp),
   filemode(fp,0),

/* Build data structure */
   init_Reflex(TotRecs,FldNames,TypeL,TxtPools),
   filepos(fp,DataFilePos,0),

/* Read all data records */
   makewindow(85,41,36," Reflex(tm) All Data Records ",0,0,25,40),
   rd_Reflexfile(TotRecs,TypeL,TxtPools,ReflexRecs),
   list_Recs(FldNames,ReflexRecs), doPrompt,
   window_attr(27),

/* Read data records sequentially */
   makewindow(85,72,33," Reflex(tm) Sequential Access ",0,40,25,40),
   filepos(fp,DataFilePos,0),
   rd_ReflexRec(TotRecs,TypeL,TxtPools,Rec),
   nl,nl,
   list_rec(FldNames,Rec),
   PressAKey,fail.
```

Accessing a dBASE III File

The first step in accessing a dBASE III file from a Turbo Prolog program is to call the tool predicate *Init_Dbase3* in the file DBASE3.PRO, which builds data structures describing dBASE III records. It has the following declaration:

```
   Init_Dbase3(REAL,FLDNAMEL,FLDDESCL)
```

in which the nonstandard **domains** are declared as follows:

```
FLDDESCL = FLDDESC*                                    /* Description for each field */
FLDDESC = FLDDESC(DBASE3TYPE,INTEGER)
DBASE3TYPE = ch;r;l;m;d

FLDNAMEL = STRING*
```

Using the tool predicate *Rd_dBase3File,* the dBASE III data records can be read and collected in a list belonging to the tool domain *DBASE3RECL,* which has the following declaration:

```
DBASE3RECL = DBASE3REC*                    /* The database is a list of records */
DBASE3REC  = DBASE3ELEM*                                /* Fields in each record */
DBASE3ELEM = char(STRING);                                         /* Characters */
             real(REAL);                             /* 64-bit IEEE floating-point */
             logical(BOOL);                                           /* Logical */
             memo(STRING);                    /* Memo text loaded from a .DBT file */
             date(STRING)                                  /* Format YYYY MM DD */

BOOL = CHAR                                            /* Y y N n T t F f or Space */
```

The record list from the domain *DBASE3RECL* and the list of field names from the domain *FLDNAMEL* make up a complete data structure describing a dBASE III (v1.1) database file.

The Toolbox file XDBASE3.DBF is a dBASE III file containing personnel data organized as follows:

Field	Type	Width
Name	Char/String	25
Birth_Date	Date	8
Salary	Numeric	8.2
Age	Numeric	2
Memo	Memo	

The corresponding Turbo Prolog record takes the form

```
[string("Frank Borland"),                                          /* Name field */
 date("19131205"),                                                 /* Birth date */
 real(10250.95),                                                       /* Salary */
 real(73),                                                                /* Age */
 memo("Frank Borland's memo")]
```

NOTE: This record is located in a list of other records.

The corresponding list of field names takes the form

```
["Name", "Birth_Date", "Salary", "Age", "Memo"]
```

Inside the toolbox, the declaration of *Rd_dBase3File* is

```
Rd_dBASE3File(REAL,FILE,FLDDESCL,DBASE3RECL)
```

The example below shows how to use this predicate to access the dBASE III file XDBASE3.DBF and the memo file XDBASE3.DBT.

```
AccessAll(DbaseRecs) :-
        openread(fp,"xdbase3.dbf"), filemode(fp,0), readdevice(fp),
        openread(mfp,"xdbase3.dbt"),filemode(mfp,0),

/* Build data structure */
        Init_Dbase3(TotRecs,FldNameL,FldDescL),

/* Read all data records */
        rd_dBase3File(TotRecs,mfp,FldDescL,RecL),

/* List all records */
        makewindow(85,41,36," dBASE III(tm) All Data Records ",0,0,25,40),
        list_recL(FldNameL,RecL), DoPrompt.
```

The two last literals (*makewindow* and *list_Recs*) are not necessary, but they display the contents of the dBASE III file in a readable manner.

Reading One dBASE III Record at a Time

The tool predicate *rd_dBase3Rec* allows you to read dBASE III records sequentially. Thus, with *rd_dBase3Rec,* it is possible to read a record, do some computation, remove the storage allocation via backtracking, and then read the next record. The following example shows how:

```
OneByeOne :-
        openread(fp,"xdbase3.dbf"), filemode(fp,0), readdevice(fp),
        openread(mfp,"xdbase3.dbt"),filemode(mfp,0),

/* Build data structure */
        Init_Dbase3(TotRecs,FldNameL,FldDescL),

/* Read records one by one */
        makewindow(85,72,33,"dBASE III(tm) Sequential Access",0,40,25,40),
        rd_dBASE3Rec(TotRecs,mfp,FldDescL,Rec),
        list_recL(FldNameL,[Rec]),fail.
```

The predicate *rd_dBase3Rec* also is non-deterministic: While there are more records in the dBASE III file, it reads the next record in the structure *Rec* and returns it to the caller. When the calling predicate fails, *rd_dBase3Rec* generates a new solution (that is, reads the next record if there is one). It has the following declaration:

```
   Rd_dBase3Rec(REAL,FILE,FLDDESCL,DBASE3REC)
```

Both predicates, *Rd_dBase3Rec* and *Rd_dBase3File,* can be seen in the following complete program, XDBASE3.PRO.

XDBASE3.PRO

```
project "xdbase3"

domains
   FILE = fp ; mfp

global predicates
   real_ints(REAL,INTEGER,INTEGER,INTEGER,INTEGER) - (o,i,i,i,i) language c

include "readext.pro"
include "dbase3.pro"

predicates
/* Listing of the database */
   list_recl(FldNameL,dBase3Recl)
   list_rec(FldNameL,dBase3Rec)
   list_elem(dBase3Elem)
   NoofNL(Integer)
   PressAKey
   doPrompt

clauses
/***********************************************************************
                        List Data from .DBF & .DBT
***********************************************************************/

   list_recL(_,[]) :- !.
   list_recL(FldNameL,[Rec|RecL]) :-
         nl,nl,
         list_rec(FldNameL,Rec), PressAkey,
         list_recL(FldNameL,RecL).

   list_rec([],[]) :- !.
   list_rec([FldName|FldNames],[Elem|Elems]) :-
         writef("\n%-12: ",FldName),
         list_elem(Elem),
         list_rec(FldNames,Elems).

   list_elem(char(Str)) :- write(Str).
   list_elem(real(Real)) :- write(Real).
   list_elem(logical(CH)) :- write(CH).
   list_elem(memo(Str)) :- write(Str).
   list_elem(date(Date)) :- write(Date).

   NoofNL(N) :- N< = 0,!.
   NoofNL(N) :- nl, N2 = N-1, NoofNL(N2).

   PressAKey :-
         makewindow(_,_,_,_,MinR,_,NoofR,_),
         cursor(R,_), R< = MinR+NoofR-8,!.

   PressAKey :- doPrompt,cursor(R,C), scroll(R,0),cursor(0,C).

   doPrompt :-
         makewindow(Nr,Att,_,_,MinR,MinC,NoofR,NoofC),
         MinR2 = MinR+NoofR-1, MinC2=MinC+NoofC/3+1,
         str_len(" Press a key",Len), Len2 = Len+1, bitxor(Att,8,Att2),
         makewindow(Nr,Att2,0,"",MinR2,MinC2,1,Len2),
         write(" Press a key"),
         readdevice(FP), readdevice(keyboard), readchar(_), readdevice(FP),
         removewindow.

/***********************************************************************
                                  Goal
***********************************************************************/
goal
   openread(fp,"xdBase3.dbf"), filemode(fp,0), readdevice(fp),
   openread(mfp,"xdBase3.dbt"),filemode(mfp,0),

/* Build data structure */
   init_dBase3(TotRecs,FldNameL,FldDescL),

/* Remember file positions */
   filepos(fp,Fposfp,0), filepos(mfp,Fposmfp,0),

/* Read all data records */
   rd_dBase3File(TotRecs,mfp,FldDescL,RecL),

/* List all records */
   makewindow(85,41,36," dBASE 3(tm) All Data Records ",0,0,25,40),
   list_recL(FldNameL,RecL), DoPrompt,
   window_attr(27),

/* Read records one by one */
   filepos(fp,Fposfp,0), filepos(mfp,Fposmfp,0),
   makewindow(85,72,33," dBASE 3(tm) Sequential Access ",0,40,25,40),
   rd_dBase3Rec(TotRecs,mfp,FldDescL,Rec),
   list_recL(FldNameL,[Rec]),fail.
```

Accessing a Lotus 1-2-3 or Symphony File

Lotus 1-2-3 and Symphony files consist of WKS and WRK records, respectively.(.WKS is the file-name extension automatically given to Lotus

1-2-3 files, and .WRK is the extension given to Symphony files.) Each record controls some aspect of the file, such as ranges and cell formats. The WKS and WRK record formats have three parts that specify an opcode, the record length, and the data. Each opcode and record length is stored as a two-byte integer, least significant byte first. The data is represented as a series of zero or more data bytes.

Since the purpose of the tool predicates in this section is to allow access to some figures from a spreadsheet, only a few types of WKS and WRK records need concern you, namely:

Opcode 0	(Version)
Length	2 bytes
Meaning	1-2-3 or Symphony version number
Data 0-1	file format version number:
	1028 is a 1-2-3 file
	1029 is a Symphony file
	1030 is 1-2-3 (v2.0) or Symphony 1.1

Opcode 13	(Integer)
Length	7 bytes
Meaning	describes integer number cell
Data 0	format byte
Data 1-2	column number
Data 3-4	row number
Data 5-6	integer value

Opcode 14	(Number)
Length	13 bytes
Meaning	defines floating-point number cell
Data 0	format byte
Data 1-2	column number
Data 3-4	row number
Data 5-12	64-bit IEEE floating-point real value

Opcode 16	(Formula)
Length	variable to 2064 bytes
Meaning	defines a formula cell
Data 0	format byte
Data 1-2	column number
Data 3-4	row number
Data 5-12	formula numeric value (64-bit IEEE floating-point format)
Data 13-n	formula definition (can be ignored in this case)

The corresponding Turbo Prolog Toolbox domain declarations take the form

```
LOTUSRECL = LOTUSREC*

LOTUSREC  = version(INTEGER);                                    /* Version number */
            elem(INTEGER,INTEGER,VALUE);

VALUE = int(INTEGER);                                       /* Integer number cell */
        real(REAL);                                            /* Real number cell */
        formula(REAL);                                     /* Defines a formula cell */
        label(STRING);                                       /* Defines a label cell */
```

If a fragment of a Symphony spreadsheet has the form

	A	B	C	D
1	2	Diff	123.654	
2	3			
3				

then the corresponding Turbo Prolog data structure would have the form

```
LotusRecL = [version(1029),                                       /* Symphony 1.0 */
             elem(0,0,int(2)),                                 /* row 1, column A */
             elem(0,1,label("Diff")),                          /* row 1, column B */
             elem(0,2,real(123.654)),                          /* row 1, column C */
             elem(1,0,int(3))]                                 /* row 2, column A */
```

Since a Lotus file can be accessed in two different ways (either by reading all cells in a single operation or by searching for a specific cell), let's deal with the two methods separately. In both cases, however, as with Reflex and dBASE III databases, there's no need to carry out any initialization. The tool predicates that handle Lotus files automatically create the necessary data structures.

Reading All Cells

The tool predicate *Rd_LotusFile* reads all cells in a spreadsheet into a data structure. It is declared as follows:

```
Rd_LotusFile(LOTUSRECL)
```

The following example shows how to read all the cells from the spreadsheet stored in the file XLOTUS.WRK on your Toolbox disk.

```
   ReadAllCells(RecL) :-
         openread(fp,"XLOTUS.wrk"), readdevice(fp), filemode(fp,0),
/* Read all cells */
         rd_LotusFile(RecL).
/* List cells */
         makewindow(85,26,36," Lotus 1-2-3 & Symphony (tm) ",0,0,25,80),
         list_recs(0,RecL).
```

Reading a Specific Cell

The tool predicate *Rd_LotusCell* searches for an instantiated cell in a spreadsheet file. The permissible flow patterns for *Rd_LotusCell* are

```
Rd_LotusCell(Rec) - (i),(o)
Rd_LotusCell(elem(Row,Col,Value))
          - (i,i,i),(i,i,o), (i,o,o), . . . , (o,o,o).
```

The following program fragment shows how to search for a cell located at row 3 in the file XLOTUS.WRK:

```
   SearchRec :-
         openread(fp,"Demo.wrk"), readdevice(fp), filemode(fp,0),
/* Read sequentially, searching for specific cell */
         Rd_LotusCell(elem(3,_,R)), write("\nCell: ",R).
```

This fragment along with the previous one, are in XLOTUS.PRO, whose listing follows.

XLOTUS.PRO

```
project "xlotus"

domains
   FILE = fp
```

```
global predicates
   real_ints(REAL,INTEGER,INTEGER,INTEGER,INTEGER) - (o,i,i,i,i) language c

include "readext.pro"
include "lotus.pro"

predicates
   list_Recs(Integer,LotusRecL)
   list_Rec(Integer,Integer,LotusRec)
   wr_Version(Integer)
   wr_elem(Value)
   NoofNL(Integer)
   PressAKey
   doPrompt

clauses
   list_Recs(_,[]) :- !.
   list_Recs(CurRow,[Rec|RecL]) :-
        list_rec(CurRow,CurRow2,Rec),
        PressAKey,
        list_Recs(CurRow2,RecL).

   list_rec(CurRow,CurRow,elem(CurRow,Col,Int)) :-
        cursor(Row,_), Col2 = Col*8, cursor(Row,Col2), wr_elem(Int).

   list_rec(CurRow,Row,elem(Row,Col,Int)) :- !,
        NL = Row-CurRow,
        NoofNL(NL), Col2 = Col*8,
        cursor(Row2,_), cursor(Row2,Col2), wr_elem(Int).

   list_rec(CurRow,CurRow,version(V)) :-
        write("\t\tVersion: "),wr_Version(V).

   wr_Version(1028) :- write("Lotus 1-2-3 1.0").
   wr_Version(1029) :- write("Symphony 1.0").
   wr_Version(1030) :- write("Lotus 1-2-3 v2.0 or Symphony 1.1").

   wr_elem(int(I)) :- writef("%8",I).
   wr_elem(real(I)) :- writef("%8",I).
   wr_elem(label(S)) :- writef("%8",S).

   NoofNL(N) :- N< = 0,!.
   NoofNL(N) :- nl, N2 = N-1, NoofNL(N2).

   PressAKey :-
        makewindow(_,_,_,_,MinR,_,NoofR,_),
        cursor(R,_), R< = MinR+NoofR-4,!.
   PressAKey :- doPrompt,cursor(R,C), scroll(R,0),cursor(0,C).

   doPrompt :-
        makewindow(Nr,Att,_,_,MinR,MinC,NoofR,NoofC),
        MinR2 = MinR+NoofR-1, MinC2 = MinC+NoofC/3+1,
        str_len(" Press a key",Len), Len2 = Len+1, bitxor(Att,8,Att2),
        makewindow(Nr,Att2,0,"",MinR2,MinC2,1,Len2),
        write(" Press a key"),
        readdevice(FP), readdevice(keyboard), readchar(_), readdevice(FP),
        removewindow.
```

```
/************************************************************************
                                                  Goal
************************************************************************/
goal
   openread(fp,"xlotus.wrk"), readdevice(fp), filemode(fp,0),

/* Read all records at once */
   rd_LotusFile(RecL),
   makewindow(85,40,36," Lotus 1-2-3 & Symphony - All Records ",
              0,0,25,80),
   write("\n\n",RecL), doPrompt,

/* Read sequentially, searching for specific record */
   filepos(fp,0,0),
   write("\n\nSearch for a record at column 0"),
   rd_Lotuscell(elem(Row,0,R)), Elem1 = elem(Row,0,R),
   write(", Record: ",Elem1),

   write("\n\nSearch for a record at row 3 and column 3"),
   rd_Lotuscell(elem(3,3,R2)), Elem2 = elem(3,3,R2),
   write(", Record: ",Elem2),
   doPrompt,

/* List records in "Spreadsheet" way */
   makewindow(85,26,36," Lotus 1-2-3 & Symphony (tm) ",0,0,25,80),
   list_recs(0,RecL).
```

C H A P T E R

7

The Parser Generator

This chapter demonstrates how to use the Toolbox's parser generator to take a grammar specified by you and automatically create a parser for that grammar. The chapter is organized in five main parts.

The first part is an overview of parsers, parser generators, and how they work. It is intended to introduce the topic to those with limited knowledge of parsers. The second part discusses how to define a grammar that conforms to the Toolbox's requirements. The third section gives two complete examples of input to and output from the parser generator, along with examples of associated scanners. The fourth section describes how to compile and use the Toolbox parser generator, and the final section discusses how to recreate it using "bootstrapping."

What Does a Parser Do?

A *parser* is a program that can recognize the underlying structure of a source text. For example, a Pascal compiler applies a parser to recognize the **if**, **while**, **repeat**, and **case** statements as well as the procedures, functions, and expressions in a Pascal source file. Parsers are used not only with programming languages but also with command interpreters to interpret user input for various other types of programs—expert system shells and natural-language interfaces, for example.

A parser translates source text into a format that is suitable for the next phase of a compiler, interpreter, or other program. The next phase in the case of a compiler, for instance, is typically code generation. In doing the translation, a parser usually performs the following sequence of steps:

1. Asks the scanner (commonly referred to as the lexical analyzer) for the next token in the input source.
2. Checks that the token is part of the legal pattern specified by the language's grammar; that is, error checking.
3. Imposes on the token, in relation to previous and/or succeeding tokens, a structure that can be used by subsequent phases of the compiler, interpreter, or other program.

The Different Types of Parsers

First, let's define some terms. A *sentence* is a list of terminal symbols. A *production* is a translation rule that you specify. A *parsing table* matches each terminal symbol with a production rule.

There are two main classes of parsers: *top-down* and *bottom-up*. Given a sentence to parse, top-down parsers start with the most general production (the start-symbol) in the grammar and try to produce a list of productions that generate that sentence.

On the other hand, bottom-up parsers start with a given sentence and work backwards to prove that it is an instance of one of the grammar's production rules. A bottom-up parser usually requires access to a parsing table. The table simply records—for each terminal symbol—which production rule the parser should use when that terminal symbol is next considered in the parser's attempt to verify the input sentence.

When working through the given sentence, a bottom-up parser looks ahead a certain amount. Otherwise it won't be able to tell, for example, whether a sentence that begins with **if** is going to be of the form **if..then..** or the form **if..then..else...**

Most parser generators produce bottom-up parsers, because it's relatively easy to generate a parsing table to drive a bottom-up parser for almost any reasonable grammar. On the other hand, you can construct a top-down parser to be general enough that each of the parsers produced by the parser generator is a special case of it. However, the initial (general) parser must be able to backtrack—a feature that is built-in to Prolog.

The Toolbox uses a top-down parser for several reasons, the main one being that it is easier to build structures during the parsing process. Also, the difference-list technique in Prolog (explained later in this chapter) makes it comparatively easy to construct a very efficient top-down parser with an arbitrary look-ahead length and with backtracking. As you will discover, the resulting parsing speed of generated parsers can be very high. If you code the scanner (lexical analyzer) in C or assembler, you could design parsers that run with nearly the same speed as Turbo Prolog.

Lexical Analysis: The Scanner

The scanner is perhaps the simplest part of any compiler. It reads the source input a character at a time, searching for the next token. The Pascal statement

```
X := Y + 1;
```

would be broken up into the tokens

```
X, :=, Y, +, 1 and ;
```

The scanner can also attach some attributes to each token. For example, the tokens X and Y in the above Pascal statement can be assigned an attribute to indicate that they are identifiers.

Describing the Grammar

The structures to be recognized by a parser are normally described by a grammar. There are many advantages to using a grammar, including:

- It gives a precise and easy to understand syntactic specification for the programs of a particular language.
- Creating a parser for a well-designed grammar can easily be automated.
- A grammar structures the input so that it is smoothly translated into object code and errors in the source input are easily detected. In the

case of the Toolbox parser generator, there is a close connection between the grammar and the domain definitions for the Turbo Prolog program (scanner) that handles the input.

A grammar can be described in several ways. The Toolbox uses a type of context-free grammar specification called Backus Naur Form (BNF) to describe the grammars the parser generator uses. Using BNF, you list the allowed productions (that is, rules) for forming valid sentences in the language specified by the grammar. Let's delve into Backus Naur Form (BNF) at this point.

Backus Naur Form Grammar

A grammar generally involves four entities: *terminals, non-terminals,* a *start symbol,* and *production rules.*

Terminals are the basic symbols from which sentences are created in a language. The word *token* is a synonym for terminal. For example, in the following sentence,

```
mary likes big cars
```

the terminals are *mary, likes, big,* and *cars*

Non-terminals are symbols that are special as far as the grammar is concerned; they denote sets of strings. For example, given the following production rule,

```
<SENTENCE> ::= <SUBJECT> likes <OBJECT>
```

the non-terminals are *<SENTENCE>*, *<SUBJECT>*, and *<OBJECT>*.

One non-terminal in the grammar is always selected as the *start symbol.* As its name implies, the start symbol is where a parser begins when determining how to parse its source input and denotes the language being defined. The non-terminal *<SENTENCE>* in the previous production rule is the start symbol.

Production rules define the ways in which grammatical structures can be built from one another and from terminals. The syntax for a production rule is

```
<non-terminal> ::= a string of non-terminals and terminals
```

An example of a production rule is

```
<SENTENCE> ::= <SUBJECT> likes <OBJECT>
               does <SUBJECT> like <OBJECT>
```

In the preceding example, two possible translations of the non-terminal *<SENTENCE>* are being defined.

Some notational conventions to keep in mind when using BNF are

1. *Non-terminals* are surrounded by < and > to easily distinguish them from *terminal* symbols; for example, *<STATEMENT>*.
2. The asterisk (*) is used in *production rules* to indicate the possibility of zero or more instances of the *non-terminal* or *terminal* symbol. For example, a language could be defined as a series of zero or more statements: `<LANGUAGE> ::= <STATEMENT>*`
3. The plus sign (+) is used in production rules to indicate the possibility of one or more instances of the *non-terminal* or *terminal* symbol. For example, an identifier in a language could be defined as

   ```
   <IDENTIFIER> ::=<LETTER>{<LETTER-OR-DIGIT>}+
   ```

   indicating that an identifier is made up of alphanumerics but must start with an alphabetic character.
4. The | mark is used to indicate "or" in a *production rule*. For example,

   ```
   <LETTER-OR-DIGIT> ::= <LETTER>|<DIGIT>
   ```

5. A *non-terminal* surrounded by [and] in a *production rule* may be used zero or one times. That is, it is optional. For example,

   ```
   <PRODBODY> ::= <PRODNAME> [ <SEPARATOR> ]
   ```

The grammar

```
<SENTENCE>  ::= <SUBJECT> likes <OBJECT>
                does <SUBJECT> like <OBJECT>
<SUBJECT>   ::= john | mary
<OBJECT>    ::= <ADJECTIVE> <NOUN>
<ADJECTIVE> ::= big | medium | small
<NOUN>      ::= books | cars
```

defines a total of six production rules: Two define the non-terminal *<SENTENCE>*, and four define each of the remaining non-terminals—*<SUBJECT>*, *<OBJECT>*, *<ADJECTIVE>*, and *<NOUN>*. The words *likes, does, like, john, mary, big, medium, small, cars,* and *books* are the terminal symbols of the language. In other words, they are the symbols from which all valid sentences are made up according to the production rules. The production rule

```
<SUBJECT> = john | mary
```

indicates that a symbol belonging to the grammatical category *<SUBJECT>* is either the word *john* or (as denoted by the | character) *mary.*

It follows that

```
mary likes big cars
```

is a valid sentence in the language described by this grammar because of the following chain of *productions*:

A *<SENTENCE>* takes the form *<SUBJECT>* likes *<OBJECT>*.
mary is a *<SUBJECT>*.
An *<OBJECT>* takes the form *<ADJECTIVE> <NOUN>*.
big is an *<ADJECTIVE>*.
cars is a *<NOUN>*.
Therefore *big cars* is an *<OBJECT>*.
Therefore *mary likes big cars* is a *<SENTENCE>*.

For a real programming-language grammar, reread the section "BNF Syntax for Turbo Prolog" in Chapter 12 of the *Turbo Prolog Owner's Handbook.* It contains a complete BNF grammar for Turbo Prolog, specified more formally than in the example above.

The Toolbox Parser Generator

A *parser generator* is a program that generates a parser from a grammar specification. Since not everyone is an expert in parser writing, a parser generator enables even non-experts to construct parsers.

The Toolbox parser generator takes a grammar specified using the BNF notation described earlier and generates a parser that can recognize sentences that conform to the grammar. After determining that a sentence conforms to the original grammar, a tree is built that shows the structure of the original source input.

Following are two subsections. The first discusses the rules that you must follow when describing a grammar for input to the Toolbox parser generator. The second section gives an example of a programming language construct, followed by its grammar specification in both BNF notation and the Toolbox's required notation. This gives you another example of the conversion process and also points out some subtleties in describing a grammar.

Specifying the Input to the Parser Generator

A formal specification of the format you must use when inputting information to the parser generator is given at the end of this section. The formal specification is in BNF format, so refer back to "Describing the Grammar" in this chapter if you have any questions about how BNF works. The next section shows an example of a grammar in standard BNF notation, followed by the equivalent code as required by the Toolbox's parser generator.

Note the distinction between the format or syntax of the parser generator's input and its formal description. BNF notation is used to describe a language that itself closely conforms to BNF notation.

First, let's follow through an informal specification in words.

Parser-generator input consists of three optional kinds of sections: user-defined predicates, user-defined domains, and production rules.

If you wish to manually code some parsing predicates, they should be declared to the parser generator at the beginning of your grammar definition, after the keywords **userdefined predicates**. Similarly, user-defined domains should be declared next, after the keywords **userdefined domains,** *before* the **predicates.**

Following **userdefined** sections, there can be any number of production-rule sections, but the keyword **productions** must precede each such section. A production begins with the name of that production in uppercase letters. There are two kind of productions: list productions and general productions.

A *list* production is a production name followed by an asterisk or a plus sign. An asterisk means zero or more occurrences of that grammatical object can appear in a valid sentence; a plus means one or more. Then, a list production can contain an optional separator specification. When no separator is given, the list simply doesn't have any separator symbols. For example, you can declare that an expression list is a (possibly empty) list of expressions separated by commas as follows:

```
EXPLIST = EXP* separator comma
```

This specification generates the following Turbo Prolog domain declaration in the domain-definition file of the parser-generator system (more details on this file later):

```
EXPLIST = EXP*
```

This domain declaration is independent of whether an asterisk or a plus is used and whether a separator is used or not.

A *general* production consists of one or more groups of individual productions, each with the same priority. The priority groups are separated by two minus signs (--), while the productions in each group are separated by a comma (,).

An *individual* production can be preceded by the keyword *rightassoc*. If the production is right associative, then its name can be followed by a (possibly empty) list of grammatical tokens. These are either names of other productions or scanner tokens. It can have an optional parameter list for when the scanner places some attributes in the token—for example, the value of an integer or the name of an identifier.

After the list of grammatical tokens comes an arrow(->), which is followed by a specification of the Prolog term that should be built from this production. This term can have zero or more arguments, the arguments being the names of productions or Turbo Prolog terms.

The production names in the list of grammatical tokens must be exactly the same, and in the same order, as the production names used in the corresponding Prolog term: The first production name in the grammatical token list must correspond to the first name in the term and so on. If two corresponding names are not equal, a conversion between the two names is inserted in the parser. For example,

```
MYPRODNAME = upper(STRING) -> MYPRODNAME
```

generates the following parsed predicate:

```
s_prodname([t(upper(STRING),_)|LL],LL,MYPRODNAME) :-!,
          STRING = MYPRODNAME.
```

The BNF grammar in Table 7.1 shows how input for the parser generator must be specified.

Table 7.1: A BNF Grammar for Parser-Generator Input

<PARSERGENERATORINPUT>	::=	**userdefined productions** <PRODNAMES>
		userdefined domains <PRODNAMES>
		productions <PRODUCTIONS>
<PRODUCTIONS>	::=	<PRODUCTION>+
<PRODUCTION>	::=	<PRODNAME> = <PRODBODY>
<PRODBODY>	::=	<PRODGROUPS>
		<PRODNAME> <STAR_PLUS> [<SEPARATOR>]
<STAR_PLUS>	::=	* | +
<SEPARATOR>	::=	<STRING>
<PRODGROUPS>	::=	<PRIORGROUP> { - - <PRIORGROUP> }*
<PRIORGROUP>	::=	<SINGPROD> { , <SINGPROD> }*
<SINGPROD>	::=	[<ASSOC>] <GRAMTOKL> -> <TERM>
<ASSOC>	::=	rightassoc
<GRAMTOKL>	::=	<GRAMTOK>*
<GRAMTOK>	::=	<PRODNAME> | <TOKK>[:CURSOR]
<TOKK>	::=	<LOWERCASESTRING> (<PRODNAMES>)
		<LOWERCASESTRING>
<TERM>	::=	<LOWERCASESTRING>
		<LOWERCASESTRING> (<PRODNAMES>)
		<PRODNAME>
<PRODNAMES>	::=	<PRODNAME> { , <PRODNAMES>}*
<PRODNAME>	::=	<UPPERCASESTRING>
<UPPERCASESTRING>	::=	{A|B|C|D|E|F|G|H|I|J|K|L|M|
		N|O|P|Q|R|S|T|U|V|W|X|Y|Z }+
<LOWERCASESTRING>	::=	{a|b|c|d|e|f|g|h|i|j|k|l|m|
		n|o|p|q|r|s|t|u|v|w|x|y|z }+

There are a few differences between the standard BNF syntax described at the beginning of the chapter and the BNF syntax that the Toolbox requires its input to be in:

- The < and > characters that normally surround a non-terminal are not allowed.
- Instead of the ::= operator to indicate the body of a production, the Toolbox uses =.

Creating Your Own Grammar

In the following pages, you'll see how to write a grammar that describes a language capable of recognizing the following construct:

```
while not finished do
   if a + 7 * f(x) > X^2 then
      write("true")
   else
      write("false");

if a > 0 then
   a = 0;
```

After describing the grammar using standard BNF notation, we'll convert it to the format required by the Toolbox.

A valid program in this programming language consists of a series of statements or sentences. These sentences may contain **while** statements, **if..then..else..** statements, **write** statements, and assignments. In the **while**, **if**, and assignment statements, expressions are used. These can contain addition, multiplication, exponentiation and relational operators, function calls, and variables and constants.

To define this language by a suitable grammar, first state that the language is a sequence of zero or more statements, each terminated by a semicolon:

```
   <LANGUAGE> ::= { <STATEMENT>; }
```

Next, describe the permissible kinds of statements:

```
   <STATEMENT> ::= while <EXP> do <STATEMENT>
                   if <EXP> then <STATEMENT> else <STATEMENT>
                   if <EXP> then <STATEMENT>
                   <IDENTIFIER> = <EXP>
                   write( <EXP> )
```

Finally, describe the expressions:

```
<EXP> ::= <EXP> <RELOP> <EXP>
          ( <EXP> )
          <IDENTIFIER>
          <STRING>
          <INTEGER>
          <EXP> + <EXP>
          <EXP> - <EXP>
          <EXP> * <EXP>
          <EXP> / <EXP>
          <EXP> ^ <EXP>
          <IDENTIFIER> ( <EXP> )
          not <EXP>
```

where

```
<RELOP> ::= > | < | = | <= | >=
<IDENTIFIER> ::= <LETTER>{ <LETTER> | <DIGIT> }*
<STRING> ::= " { <LETTER> }* "
<INTEGER> ::= <DIGIT>{ <DIGIT> }*
<LETTER> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|
             a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z| _
<DIGIT> ::= 0|1|2|3|4|5|6|7|8|9
```

Note that there are several deficiencies in the above grammar. Most notably, there are no provisions for a signed integer and no characters like punctuations and so forth in strings. Also note that the above grammar is described using BNF notation. It does *not* conform to the rules that were described in the last section. The following is the Toolbox version of the grammar and can be given directly to the Toolbox parser generator:

```
productions
   EXP =     EXP plus EXP      -> plus(EXP,EXP),
             EXP minus EXP     -> minus(EXP,EXP)
             --
             EXP mult EXP      -> mult(EXP,EXP),
             EXP div EXP       -> div(EXP,EXP)
             --
             rightassoc EXP power EXP -> power(EXP,EXP)
             --
             EXP less EXP      -> less(EXP,EXP),
             EXP greater EXP   -> greater(EXP,EXP),
             EXP equal EXP     -> equivalent(EXP,EXP),
             EXP greater_equal EXP -> greater_equal(EXP,EXP),
             EXP less_equal EXP -> less_equal(EXP,EXP)
             --
             id(STRING) lpar EXP rpar
                               -> call(STRING,EXP),
             int(INTEGER)      -> int(INTEGER),
             str(STRING)       -> str(STRING),
             lpar EXP rpar     -> EXP,
             not EXP           -> not(EXP)
```

```
STATEMENT = if_ EXP then STATEMENT else STATEMENT
                        -> ifthenelse(EXP,STATEMENT,STATEMENT),
            if_ EXP then STATEMENT
                        -> ifthen(EXP,STATEMENT),
            while EXP do STATEMENT
                        -> while(EXP,STATEMENT),
            id(STRING) equal EXP
                        -> assignment(STRING,EXP),
            write_ lpar STRING rpar
                        -> write(STRING)

LANGUAGE = STATEMENT* separator semicolon
```

There were two primary changes made in converting the grammar above from straight BNF to the Toolbox's format. First, cosmetic changes were needed to remove the < and > surrounding *non-terminals* and change ::= to =. Then, the grammar had to show how a Turbo Prolog term should be associated with each defining production in the source language grammar.

Notice that **if** represented as a scanner token is *if_* and that **write** becomes *write_*. This is a useful habit to adopt to avoid confusion between Turbo Prolog keywords and scanner tokens. Likewise, the tokenized forms of (and) are *lpar* and *rpar* respectively. Also, integer values in the source text are given in terms of the functor *int(..)* and identifiers in terms of *id(..)*.

Two complete examples of how the Toolbox parser generator works are provided in the next section of this chapter. After studying these complete examples, input the grammar given above to the parser generator, develop an appropriate scanner for the grammar and test the system with the above statements as input.

As mentioned previously, some *terminal* symbols carry an attribute along with them. For example, the terminal *<IDENTIFIER>* carries the name of the identifier and the terminal *<INTEGER>* carries the value of the integer.

There are many issues connected with such a grammar, including:

- *Precedence*: Addition and multiplication are normally given different priorities so that multiplication binds tighter than addition. For example, when evaluating the expression 2 + 3 * 4, the result of 3 * 4 is normally evaluated first. Priorities like this are normally reflected in the grammar.
- *Associativity*: In an expression like 2 + 3 + 4, the addition operator is usually assumed to be left associative. The operation 2 + 3 is carried out first and not 3 + 4. In the expression 2^3^4, the operation 3^4 should be carried out first.

- *Ambiguity*: There is often more than one way to combine the input to a parser into a grammatical structure. The **if..then..else..** sentence is the most common example. In the context of the above grammar, consider the construction:

```
if a then
if b then sent1
else sent2
```

- The grammar does not specify whether the construction should be interpreted as

```
if a then ( if b then sent1 else sent2 )
```

or as

```
if a then ( if b then sent1) else sent2
```

In the Toolbox parser generator, these problems have been resolved by the putting the following constraints on how the source-language grammar should be specified:

- *Precedence*: Productions for one production name should be separated into groups having the same priority. Thus, plus (+) and minus (–) should share the same group; multiplication (X) and division (/) should belong to another.
- *Associativity*: Operators are, by default, left associative. If they are right associative like the exponentiation operator, they must be preceded by the Toolbox keyword *rightassoc.*
- *Ambiguity*: In considering the order of the productions, the parser chooses productions appearing earlier in the production list in preference to those appearing later. In the example above, according to the grammar specified, the **if** statement with the **else** clause comes before the **if** statement without an **else**. Therefore, the second of the two interpretations above will be the one the parser chooses. In other words, the order in which productions are defined in the grammar dictates the priority by which the production rules are used.

Examples Demonstrating the Parser Generator

This section gives two complete examples of how to use the parser generator. The first example is introduced with a demonstration of how a grammar would typically be represented using the Toolbox's specifications.

The second example is preceded by a discussion of how to incorporate error handling into your grammar/parser.

After reading through the examples, use Turbo Prolog to create the parser generator PARSER.EXE and experiment with the examples.

There are two important things to keep in mind while reading the following sections. First, for every grammar given as input, the Toolbox parser generator produces a parser similar to the file XPARS.PAR in basic structure. The program is complicated only by considerations of priority, associativity, efficiency, and error detection. Second, in every case you must write the scanner that tokenizes the source language input so it is suitable for use by the automatically generated parser. Use the scanner from the second example, XMINIGOL.SCA, to make this task easier. Only minor modifications should be necessary to make it work with a different grammar.

The Toolbox Version of a Grammar

The input to the Toolbox parser generator must show how a Turbo Prolog term should be associated with each defining production in the source-language grammar. Thus, given the following source language statement as input,

```
if A < 0 then B = 4 else B = 3 * A
```

or its equivalent as a list of tokens output from the scanner,

```
[if_,id("A"),lessthan,int(0),then,id("B"),becomes,int(4),
 else,id("B"),becomes,int(3),mult,id("A")]
```

you should require the generated parser to output the Prolog term:

```
ifthenelse(less(id("A"),int(0)),becomes(id("B"),int(4)),
          becomes(id("B"),mult(int(3),id("A"))))
```

Rather than do this for every possible sentence of the source language, base the specification of your requirements on the grammar of the source language. Suppose that you wish to generate a parser for (arithmetic) expressions that satisfy the following grammar (note that the description of this grammar conforms closely to the Toolbox's grammar specification rules):

```
EXPRESSION = EXPRESSION plus EXPRESSION
EXPRESSION minus EXPRESSION
--
EXPRESSION mult EXPRESSION
EXPRESSION div EXPRESSION
--
rightassoc EXPRESSION power EXPRESSION
--
id(STRING)
int(INTEGER)
```

This says that an EXPRESSION can take one of seven different forms, with five of the forms containing sub-expressions. Thus, each of the following is a sentence in the above grammar:

```
id("AMOUNT")
int(456)
int(23) mult id("COST")
id("JANSALES") plus id("FEBSALES") minus id("ADCOSTS")
id(M) mult id(C) power int(2)
```

Notice that the grammar is given in a form that uses language elements as they would appear when output from the scanner. In particular, identifiers and integer values have to be attached to the functors *id* and *int*. The two minus signs (--) separate defining productions for the grammar that have the same priority, and the keyword *rightassoc* indicates a production that is right associative.

Example of the Parser Generator's Input Grammar

Your input to the parser generator has to show how a Turbo Prolog term should be associated with each defining production in the source-language grammar. Just as the BNF grammar had to be converted to the Toolbox format in the previous section, you must do the same for the grammar defined here. This is shown in Table 7.1; everything inside the central box must be given to the parser generator as input. The grammar given below is contained in XPARS.GRM.

Table 7.2: Corresponding Terms for Grammar Production

Grammar as Scanned Tokens	Turbo Prolog Term from Generated Parser

```
productions
   EXP = EXP plus EXP              -> plus(EXP,EXP),
         EXP minus EXP             -> minus(EXP,EXP)
         --
         EXP mult EXP              -> mult(EXP,EXP),
         EXP div EXP               -> div(EXP,EXP)
         --
         rightassoc EXP power EXP  -> power(EXP,EXP)
         --
         id(STRING)                -> id(STRING),
         int(INTEGER)              -> int(INTEGER)
```

Notice the arrow (->) between the source-language grammar production and its corresponding Prolog term. Although the functors *plus(), minus(),* and so on are given the same names as the scanner token from which they derive, this is not mandatory. The user has to implement these clauses in the program that uses the parser.

Example of a Parser Domains File Generated by the Parser

The grammar specification in Table 7.2 generates the domain definitions (shown in Table 7.2) in the parser generator's domain-definition file. These domain definitions are in the file XPARS.DOM.

XPARS.DOM

```
/*********************************************************************
                         Domain definitions
*********************************************************************/

domains
   EXP = plus(EXP,EXP);
         minus(EXP,EXP);
         mult(EXP,EXP);
```

```
        div(EXP,EXP);
        power(EXP,EXP);
        id(STRING);
        int(INTEGER)

   TOK = plus();
        minus();
        mult();
        div();
        power();
        id(STRING);
        int(INTEGER);
        nill
```

Example of a Scanner for the Generated Parser

An example of a scanner for the simple programming language grammar given in Table 7.1 is in the file XPARS.SCA. The program is constructed using the Turbo Prolog standard predicate *fronttoken*.

XPARS.SCA

```
/* Remove the comments around the include directive below in */
/* order to run XPARS.SCA as a standalone program. */

/* include "xpars.dom" */

domains
   ttok = t(tok,integer)
   tokl = ttok*

predicates
   tokl(string,tokl)
   maketok(string,tok,string,string)
   str_tok(string,tok)

clauses
   tokl(STR,[t(TOK,0)|TOKL]) :-
        fronttoken(STR,STRTOK,STR1),!,
        maketok(STRTOK,TOK,STR1,STR2),
        tokl(STR2,TOKL).
   tokl(_,[]).

   str_tok("+",plus) :- !.
   str_tok("-",minus) :- !.
   str_tok("*",mult) :- !.
   str_tok("/",div) :- !.
   str_tok("^",power) :- !.
   maketok(STR,TOK,S,S) :- str_tok(STR,TOK),!.
   maketok(INTSTR,int(INTEGER),S,S) :- str_int(INTSTR,INTEGER),!.
   maketok(STRING,id(STRING),S,S) :- isname(STRING),!.
```

If you remove the comment marks around the include directive, as mentioned in the source code to the scanner, you can run it as a standalone program. When the scanner is run and given the goal

```
   tokl("1/A+34*2",A).
```

the result will be

```
   A = [t(int(1),0),t(div,0),t(id("A"),0),t(plus,0),
        t(int(34),0),t(mult,0),t(int(2),0)]
   1 Solution
```

Notice that the tokenized forms of *(* and *)* are *lpar* and *rpar,* respectively. This is a useful habit to adopt in order to avoid confusion between Turbo Prolog keywords and scanner tokens. Also, integer values in the source text are given in terms of the functor *int(..)* and identifiers in terms of *id(..)*.

Example of Generated Parser

XPARS.PAR contains the parser that was generated by the Toolbox parser generator to recognize the above sentence as conforming to the example grammar given in Table 7.1. Its contents are given below:

XPARS.PAR

```
/*********************************************************************
                         Parsing predicates
*********************************************************************/

predicates
   s_exp(TOKL,TOKL,EXP)
   s_expl(TOKL,TOKL,EXP)
   s_exp5(TOKL,TOKL,EXP,EXP)
   s_exp2(TOKL,TOKL,EXP)
   s_exp6(TOKL,TOKL,EXP,EXP)
   s_exp3(TOKL,TOKL,EXP)
   s_exp7(TOKL,TOKL,EXP,EXP)
   s_exp4(TOKL,TOKL,EXP)

clauses
   s_exp(LL1,LL0,EXP) :-
         s_expl(LL1,LL0,EXP).
```

```
s_exp1(LL1,LL0,EXP_) :-
      s_exp2(LL1,LL2,EXP),
      s_exp5(LL2,LL0,EXP,EXP_).

s_exp2(LL1,LL0,EXP_) :-
      s_exp3(LL1,LL2,EXP),
      s_exp6(LL2,LL0,EXP,EXP_).

s_exp3(LL1,LL0,EXP_) :-
      s_exp4(LL1,LL2,EXP),
      s_exp7(LL2,LL0,EXP,EXP_).

s_exp4([t(id(STRING),_)|LL],LL,id(STRING)) :- !.
s_exp4([t(int(INTEGER),_)|LL],LL,int(INTEGER)) :- !.
s_exp4(LL,_,_) :- syntax_error(exp4,LL),fail.

s_exp5([t(plus,_)|LL1],LL0,EXP,EXP_) :- !,
      s_exp2(LL1,LL2,EXP1),
      s_exp5(LL2,LL0,plus(EXP,EXP1),EXP_).
s_exp5([t(minus,_)|LL1],LL0,EXP,EXP_) :- !,
      s_exp2(LL1,LL2,EXP1),
      s_exp5(LL2,LL0,minus(EXP,EXP1),EXP_).
s_exp5(LL,LL,EXP,EXP).

s_exp6([t(mult,_)|LL1],LL0,EXP,EXP_) :- !,
      s_exp3(LL1,LL2,EXP1),
      s_exp6(LL2,LL0,mult(EXP,EXP1),EXP_).
s_exp6([t(div,_)|LL1],LL0,EXP,EXP_) :- !,
      s_exp3(LL1,LL2,EXP1),
      s_exp6(LL2,LL0,div(EXP,EXP1),EXP_).
s_exp6(LL,LL,EXP,EXP).

s_exp7([t(power,_)|LL1],LL0,EXP,power(EXP,EXP1)) :- !,
      s_exp3(LL1,LL0,EXP1).
s_exp7(LL,LL,EXP,EXP).
```

Example of the Scanner and Generated Parser Used Together

The generated parser can't be run as a standalone program without several modifications. Below is an example program that incorporates both the scanner and the parser:

XPARS.PRO

```
include "xpars.dom"                    /* Domain declarations for the parser */

include "xpars.sca"                    /* scanner called by the parser */
```

```
predicates                                          /* The generated parser requires */
   syntax_error(STRING,TOKL)                          /* this predicate to be declared */

include "xpars.par"                                 /* Parser created by the Toolbox */

clauses
   syntax_error(_,_) :- write("No error detection").
```

Try running the program and giving it the same goal previously given to the scanner:

```
goal: tokl("1/A+34*2",Tok_list).
```

As expected, the result is

```
   Tok_list = [t(int(1),0),t(div,0),t(id("A"),0),t(plus,0),
               t(int(34),0),t(mult,0),t(int(2),0)]
   1 Solution
```

Now take the output generated by the scanner (the tokenized version of the original source string) and give it directly to the parser:

```
goal:  s_exp([t(int(1),0),t(div,0),t(id("A"),0),
       t(plus,0),t(int(34),0),t(mult,0),t(int(2),0)],_,EXP).
```

The result of this **goal** is

```
   EXP = plus(div(int(1),id("A")),mult(int(34),int(2)))
   1 Solution
```

So, *EXP* represents the final output from the parser.

Each parsing predicate takes something off the list input in its first parameter and returns a shorter list in its second parameter. The presence of these two lists has lead to the term "parsing with difference lists" for this technique. If you are unfamiliar with it, run the program with trace turned on.

As a final example, try the **goal**

```
goal tokl("1/A+34*2",Tok_list),s_exp(Tok_list,_,EXP).
```

Here the scanner has been told to tokenize the input string and then pass it on to the parser. The solution is, therefore, a combination of the solutions to the previous examples:

```
Tok_list = [t(int(1),0),t(div,0),t(id("A"),0),t(plus,0),
            t(int(34),0),t(mult,0),t(int(2),0)],

EXP = plus(div(int(1),id("A")),mult(int(34),int(2)))
1 Solution
```

Error Handling

The input for the Toolbox parser generator described in Table 7.1 is fairly typical, except for coping with the complications of error trapping.

If there are type errors in the source-language input to the generated parser, or if one of the variables referred to does not have a value, you can set up the generated parser to point to the offending token in the source text. For this purpose, it is possible to obtain the source-text cursor position of each of the (source-language grammar) terminal symbols by appending *:CURSOR* to the relevant scanner token. For example,

```
LABEL = id(STRING):CURSOR -> id(STRING,CURSOR)
```

specifies that you want to record source-text positions of all *LABEL*s. By a straightforward extension of this idea, it is possible to record more than one cursor position in a single production:

```
RANGE = int(INTEGER):CURSOR rangesep int(INTEGER):CURSOR ->
        range(INTEGER,CURSOR,INTEGER,CURSOR)
```

Notice that domains to the left and right of the → should also agree here.

Suppose your source language comprises only **if..then..else..** statements, in which the logical expression can consist only of integer values and the < sign. Suppose also that only assignment statements can follow the **then** and **else** parts. Then the input to the parser generator could take the form

```
SENTENCE = if_ EXP then ASSIGNMENT else ASSIGNMENT ->
           ifthenelse(EXP,ASSIGNMENT,ASSIGNMENT)

ASSIGNMENT = id(STRING) equal EXP -> assignment(STRING, EXP)

EXP = int(INTEGER) lessthan int(INTEGER) -> logical(INTEGER,INTEGER)
```

assuming that the tokenized forms of = and < are *equal* and *lessthan*, respectively.

Now, let's change this so that the parser generator produces code to keep track of the cursor position for the identifier involved:

```
SENTENCE = if_ EXP then ASSIGNMENT else ASSIGNMENT ->
           ifthenelse(EXP,ASSIGNMENT,ASSIGNMENT)

ASSIGNMENT = id(STRING):CURSOR equal EXP -> assignment(STRING,CURSOR,EXP)

EXP = int(INTEGER) lessthan int(INTEGER) -> logical(INTEGER,INTEGER)
```

Next, consider function assignments of the form

```
f(X) = an expression involving identifiers, multiplication, and exponentiation
```

Let's give a specification that allows you to keep track of the cursor:

```
FUNCDEF = id(STRING):CURSOR lpar id(STRING):CURSOR rpar equal EXP ->
          function(STRING,CURSOR,STRING,CURSOR)

EXP = id(STRING):CURSOR -> id(STRING,CURSOR)
      EXP mult EXP -> mult(EXP,EXP)
      rightassoc EXP power EXP -> power(EXP,EXP)
      id(STRING):CURSOR lpar EXP rpar -> call(STRING,CURSOR,EXP)
```

Using the Whole System in a Working Sample

Following is another example that implements a parser for an ALGOL-like language, called Minigol. This grammar is in the file XMINIGOL.GRM. The files used in this example can form the basis for further experiments with grammars for different languages. First, give the grammar for the parser generator to generate a Minigol parser:

```
userdefined domains
   PROCID

productions
   EXP =      EXP plus EXP           -> plus(EXP,EXP),
              EXP minus EXP          -> minus(EXP,EXP)
              --
              EXP mult EXP           -> mult(EXP,EXP),
              EXP div EXP            -> div(EXP,EXP)
              --
              rightassoc EXP power EXP
                                     -> power(EXP,EXP),
              EXP exclmmark          -> factorial(EXP),
              EXP questionmark EXP colon EXP
                                     -> conditional(EXP,EXP,EXP)
              --
              id(STRING) lpar PARMLIST rpar
                                     -> call(PROCID,PARMLIST),
              id(STRING)             -> var(STRING),
              minus EXP              -> neg(EXP),
              int(INTEGER)           -> int(INTEGER),
              real(REAL)             -> real(REAL),
              str(STRING)            -> str(STRING),
          char(CHAR)              -> char(CHAR),
          lpar EXP rpar           -> EXP

PARMLIST = EXP+ separator comma

SENT =    if_ EXP then SENT else SENT    -> ifthenelse(EXP,SENT,SENT),
          if_ EXP then SENT              -> ifthen(EXP,SENT),
          while EXP do SENT              -> while(EXP,SENT),
          goto int(INTEGER)              -> goto_line(INTEGER),
          goto id(STRING)                -> goto_lbl(STRING)
```

Now, put the file XMINIGOL.GRM through the parser generator to create the files XMINIGOL.DOM and XMINIGOL.PAR.

The Generated Domain Definitions

XMINIGOL.DOM contains the domain definitions that the generated parser and the scanner (which you have yet to write) need to use. Check that the code is as follows:

```
/*******************************************************************
                      Domain definitions
*******************************************************************/

domains
  EXP =          plus(EXP,EXP);
                 minus(EXP,EXP);
                 mult(EXP,EXP);
                 div(EXP,EXP);
                 power(EXP,EXP);
                 factorial(EXP);
                 conditional(EXP,EXP,EXP);
                 call(PROCID,PARMLIST);
                 var(STRING);
                 neg(EXP);
                 int(INTEGER);
                 real(REAL);
                 str(STRING);
                 char(CHAR)

  PARMLIST =     EXP*

  SENT =         ifthenelse(EXP,SENT,SENT);
                 ifthen(EXP,SENT);
                 while(EXP,SENT);
                 goto_line(INTEGER);
                 goto_lbl(STRING)

  TOK =          plus();
                 minus();
                 mult();
                 div();
                 power();
```

```
                    exclmmark();
                    questionmark();
                    colon();
                    id(STRING);
                    lpar();
                    rpar();
                    int(INTEGER);
                    real(REAL);
                    str(STRING);
                    char(CHAR);
                    if_();
                    then();
                    else();
                    while();
                    do();
                    goto();
                    comma;
                    nil
```

The Generated Parser

XMINIGOL.PAR contains the generated parser. Just glance through it for now. (Later, after you've understood the underlying structure of the parsing predicates, take a more detailed look and make your own modifications.)

```
/******************************************************************
                    Parsing predicates
******************************************************************/

predicates
   s_parmlist(TOKL,TOKL,PARMLIST)
   s_parmlist1(TOKL,TOKL,PARMLIST)
   s_exp(TOKL,TOKL,EXP)
   s_exp1(TOKL,TOKL,EXP)
   s_exp5(TOKL,TOKL,EXP,EXP)
   s_exp2(TOKL,TOKL,EXP)
   s_exp6(TOKL,TOKL,EXP,EXP)
   s_exp3(TOKL,TOKL,EXP)
   s_exp7(TOKL,TOKL,EXP,EXP)
   s_exp4(TOKL,TOKL,EXP)
   s_exp8(TOKL,TOKL,STRING,EXP)
   s_sent(TOKL,TOKL,SENT)

clauses
   s_exp(LL1,LL0,EXP) :-
        s_exp1(LL1,LL0,EXP).

   s_exp1(LL1,LL0,EXP_) :-
        s_exp2(LL1,LL2,EXP),
        s_exp5(LL2,LL0,EXP,EXP_).

   s_exp2(LL1,LL0,EXP_) :-
        s_exp3(LL1,LL2,EXP),
          s_exp6(LL2,LL0,EXP,EXP_).

     s_exp3(LL1,LL0,EXP_) :-
          s_exp4(LL1,LL2,EXP),
          s_exp7(LL2,LL0,EXP,EXP_).

     s_exp4([t(id(STRING),_)|LL1],LL0,EXP_) :- !,
          s_exp8(LL1,LL0,STRING,EXP_).
     s_exp4([t(minus,_)|LL1],LL0,neg(EXP)) :- !,
          s_exp4(LL1,LL0,EXP).
     s_exp4([t(int(INTEGER),_)|LL],LL,int(INTEGER)) :- !.
     s_exp4([t(real(REAL),_)|LL],LL,real(REAL)) :- !.
     s_exp4([t(str(STRING),_)|LL],LL,str(STRING)) :- !.
     s_exp4([t(char(CHAR),_)|LL],LL,char(CHAR)) :- !.
     s_exp4([t(lpar,_)|LL1],LL0,EXP) :- !,
          s_exp(LL1,LL2,EXP),
          expect(t(rpar,_),LL2,LL0).
     s_exp4(LL,_,_) :- syntax_error(exp4,LL),fail.

     s_sent([t(if_,_)|LL1],LL0,ifthenelse(EXP,SENT,SENT1)) :-
          s_exp(LL1,LL2,EXP),
          expect(t(then,_),LL2,LL3),
          s_sent(LL3,LL4,SENT),
          expect(t(else,_),LL4,LL5),
          s_sent(LL5,LL0,SENT1),!.
     s_sent([t(if_,_)|LL1],LL0,ifthen(EXP,SENT)) :- !,
          s_exp(LL1,LL2,EXP),
          expect(t(then,_),LL2,LL3),
          s_sent(LL3,LL0,SENT).
     s_sent([t(while,_)|LL1],LL0,while(EXP,SENT)) :- !,
          s_exp(LL1,LL2,EXP),
          expect(t(do,_),LL2,LL3),
          s_sent(LL3,LL0,SENT).
     s_sent([t(goto,_)|LL1],LL0,goto_line(INTEGER)) :-
          expect(t(int(INTEGER),_),LL1,LL0),!.
     s_sent([t(goto,_)|LL1],LL0,goto_lbl(STRING)) :- !,
          expect(t(id(STRING),_),LL1,LL0).
     s_sent(LL,_,_) :- syntax_error(sent,LL),fail.

     s_exp5([t(plus,_)|LL1],LL0,EXP,EXP_) :- !,
          s_exp2(LL1,LL2,EXP1),
          s_exp5(LL2,LL0,plus(EXP,EXP1),EXP_).
     s_exp5([t(minus,_)|LL1],LL0,EXP,EXP_) :- !,
          s_exp2(LL1,LL2,EXP1),
          s_exp5(LL2,LL0,minus(EXP,EXP1),EXP_).
     s_exp5(LL,LL,EXP,EXP).

     s_exp6([t(mult,_)|LL1],LL0,EXP,EXP_) :- !,
          s_exp3(LL1,LL2,EXP1),
          s_exp6(LL2,LL0,mult(EXP,EXP1),EXP_).
     s_exp6([t(div,_)|LL1],LL0,EXP,EXP_) :- !,
          s_exp3(LL1,LL2,EXP1),
          s_exp6(LL2,LL0,div(EXP,EXP1),EXP_).
     s_exp6(LL,LL,EXP,EXP).

     s_exp7([t(power,_)|LL1],LL0,EXP,power(EXP,EXP1)) :- !,
          s_exp3(LL1,LL0,EXP1).
     s_exp7([t(exclmmark,_)|LL],LL,EXP,factorial(EXP)) :- !.
     s_exp7([t(questionmark,_)|LL1],LL0,EXP,EXP_) :- !,
        s_exp4(LL1,LL2,EXP1),
        expect(t(colon,_),LL2,LL3),
        s_exp4(LL3,LL4,EXP2),
        s_exp7(LL4,LL0,conditional(EXP,EXP1,EXP2),EXP_).
    s_exp7(LL,LL,EXP,EXP).

    s_exp8([t(lpar,_)|LL1],LL0,STRING,call(PROCID,PARMLIST)) :- !,
        s_parmlist(LL1,LL2,PARMLIST),
        expect(t(rpar,_),LL2,LL0),STRING = PROCID.
    s_exp8(LL,LL,STRING,var(STRING)) :- !.

    s_parmlist(LL1,LL0,[EXP|PARMLIST]) :-
        s_exp(LL1,LL2,EXP),
        s_parmlist1(LL2,LL0,PARMLIST).

    s_parmlist1([t(comma,_)|LL1],LL2,PARMLIST) :- !,
        s_parmlist(LL1,LL2,PARMLIST).
    s_parmlist1(LL,LL,[]).
```

The Scanner

It's easier to try out the parser once you've implemented a scanner for Minigol. What you need is a program that examines a Minigol source text and tokenizes it into the tokens given in XMINIGOL.DOM.

The file XMINIGOL.SCA contains such a program, although five lines have been commented out so that it can be incorporated into the larger program in the next section. If you wish to run the scanner in standalone mode, it is necessary to uncomment these five lines.

XMINIGOL.SCA

```
domains

/* CURSOR = integer */
/* PROCID = STRING */
    CURSORTOK = t(TOK,CURSOR)
    TOKL = CURSORTOK*

/* include "XMINIGOL.DOM" */

predicates

/* scan_error(STRING,CURSOR) */
    tokl(CURSOR,STRING,TOKL)
    maketok(CURSOR,STRING,TOK,STRING,STRING,CURSOR)
    str_tok(STRING,TOK)
    scan_str(CURSOR,STRING,STRING,STRING)
    search_ch(CHAR,STRING,INTEGER,INTEGER)
```

```
   skipspaces(STRING,STRING,INTEGER)
   isspace(CHAR)

clauses

/* scan_error(_,_) :- write("No error recovery"). */
   tokl(POS,STR,[t(TOK,POS1)|TOKL]) :-
         skipspaces(STR,STR1,NOOFSP),
         POS1 = POS+NOOFSP,
         fronttoken(STR1,STRTOK,STR2),!,
         maketok(POS,STRTOK,TOK,STR2,STR3,LEN1),
         str_len(STRTOK,LEN),
         POS2 = POS1+LEN+LEN1,
         tokl(POS2,STR3,TOKL).
   tokl(_,_,[]).

   skipspaces(STR,STR2,NOOFSP) :-
         frontchar(STR,CH,STR1),isspace(CH),!,
         skipspaces(STR1,STR2,N1),
         NOOFSP = N1+1.
   skipspaces(STR,STR,0).

   isspace(' ').
   isspace('\t').
   isspace('\n').

   str_tok(",",comma) :- !.
   str_tok("+",plus) :- !.
   str_tok("-",minus) :- !.
   str_tok("*",mult) :- !.
   str_tok("/",div) :- !.
   str_tok("^",power) :- !.
   str_tok("?",questionmark) :- !.
   str_tok(":",colon) :- !.
   str_tok("!",exclmmark) :- !.
   str_tok("(",lpar) :- !.
   str_tok(")",rpar) :- !.
   str_tok("if",if_) :- !.
   str_tok("then",then) :- !.
   str_tok("else",else) :- !.
   str_tok("while",while) :- !.
   str_tok("do",do) :- !.
   str_tok("goto",goto) :- !.

   maketok(_,STR,TOK,S,S,0):-str_tok(STR,TOK),!.
   maketok(_,"'",char(T),S1,S3,2) :- !,frontchar(S1,T,S2),frontchar(S2,_,S3).
   maketok(CURSOR,"\"",str(STR),S1,S2,LEN) :- !,
         scan_str(CURSOR,S1,S2,STR),str_len(STR,LEN1),LEN = LEN1+1.
   maketok(_,INTSTR,int(INTEGER),S,S,0) :- str_int(INTSTR,INTEGER),!.
   maketok(_,REALSTR,real(REAL),S,S,0) :- str_real(REALSTR,REAL),!.
   maketok(_,STRING,id(STRING),S,S,0) :- isname(STRING),!.
   maketok(CURSOR,_,_,_,_,_) :- scan_error("Illegal token",CURSOR),fail.

   scan_str(_,IN,OUT,STR) :-
         search_ch('"',IN,0,N),
         frontstr(N,IN,STR,OUT1),!,
         frontchar(OUT1,_,OUT).
   scan_str(CURSOR,_,_,_) :- scan_error("String not terminated",CURSOR),fail.
```

```
search_ch(CH,STR,N,N) :-
      frontchar(STR,CH,_),!.
search_ch(CH,STR,N,N1) :-
      frontchar(STR,_,S1),
      N2 = N+1,
      search_ch(CH,S1,N2,N1).
```

If you remove the comments and run the scanner with the goal

```
goal: tokl(0,"if a+2*f(7) then goto label1
        else while bool goto 7",Tok_list).
```

the result is

```
Tok_list = [t(if_,0),t(id("a"),3),t(plus,4),t(int(2),5),t(mult,6),t(id("f"),7),
            t(lpar,8),t(int(7),9),t(rpar,10),t(then,12),t(goto,17),
            t(id("label1"),22),t(else,29),t(while,34),t(id("bool"),40),
            t(goto,45),t(int(7),50)]
```

The main file, XMINIGOL.PRO, contains a program that gives a framework for implementing a compiling system like the Turbo Prolog system itself, but in which Minigol is the language used. This program provides a pull-down menu with the options Files, Edit, and Compile. An editor window is used to create and edit the source-language text. Once some Minigol statements have been typed in, they are given to the Minigol parser automatically, and the compiled result is displayed on the screen. Try reentering the Minigol source text

```
if a+2*f(7)+1 then goto label1 else while bool do goto 7
```

to this friendlier set-up, and verify that the result is

```
ifthen(plus(var("a"),mult(int(2),call("f",[int(7)]))),
       goto_lbl("label1"))
```

XMINIGOL.PRO

```
check_determ code=4000

include "tdoms.pro"

domains
   CURSOR = INTEGER
   PROCID = STRING

database
   source(STRING)
   filename(STRING)
   pdwstate(ROW,COL,SYMBOL,ROW,COL)
   insmode
   lineinpstate(STRING,COL)
   lineinpflag
   error(STRING,CURSOR)

include "tpreds.pro"
include "status.pro"
include "pulldown.pro"
include "lineinp.pro"
include "filename.pro"

include "xminigol.dom"

predicates
   scan_error(STRING,CURSOR)

include "xminigol.sca"

predicates
   strlist_str(STRINGLIST,STRING)
   ed(STRING,CURSOR)
   change(DBASEDOM)
   resetflags
   nondeterm repparse
   parse
   better_error(CURSOR)
   new_error(STRING,CURSOR)

predicates
   expect(CURSORTOK,TOKL,TOKL)
   syntax_error(STRING,TOKL)
   checkempty(TOKL)

include "xminigol.par"

clauses
   better_error(CURSOR) :-
         error(_,OLDCURSOR),OLDCURSOR >= CURSOR,!,fail.
   better_error(_).

   new_error(_,_) :- retract(error(_,_)),fail.
   new_error(MSG,CURSOR) :- assertz(error(MSG,CURSOR)).

   expect(TOK,[TOK|L],L) :- !.
   expect(t(TOK,_),[t(_,CURSOR)|_],_) :-
         better_error(CURSOR),
         str_tok(STR,TOK),
         concat(STR," expected",MSG),
         new_error(MSG,CURSOR),fail.

   syntax_error(PROD,[t(_,CURSOR)|_]) :-
         better_error(CURSOR),
         concat("Syntax error in ",PROD,MSG),
         new_error(MSG,CURSOR),fail.

   scan_error(MSG,CURSOR) :- ed(MSG,CURSOR),fail.
```

```
checkempty([]) :- !.
checkempty([t(_,CURSOR)|_]) :-
      better_error(CURSOR),
      new_error("Syntax error",CURSOR).

strlist_str([],"").
strlist_str([H|T],STR) :-
      concat(H," ",H1),
      strlist_str(T,STR1),
      concat(H1,STR1,STR).

ed(MSG,CURSOR) :-
      source(TXT),
      editmsg(TXT,TXT1,"","",MSG,CURSOR,"",RET),
      RET><1,!,
      change(source(TXT1)).

resetflags :- retract(error(_,_)),fail.
resetflags.

change(source(_)) :- retract(source(_)),fail.
change(filename(_)) :- retract(filename(_)),fail.
change(X) :- assertz(X).

repparse.
repparse :- error(MSG,CURSOR),ed(MSG,CURSOR),!,repparse.

parse :-
      repparse,
      resetflags,
      source(STR1),
      tokl(0,STR1,L),
      s_sent(L,L1,X),
      checkempty(L1),!,
      makewindow(2,23,23,"",0,0,25,80),
      write(X),
      readkey(_),
      removewindow.
parse :-
      write(">> Parsing aborted"),nl,beep.

pdwaction(1,0) :-
      shiftwindow(OLD),shiftwindow(1),
      parse,
      shiftwindow(OLD),
      refreshstatus.
pdwaction(2,0) :-
      shiftwindow(OLD),shiftwindow(1),
      source(TXT),
      editmsg(TXT,TXT1,"","","",0,"",RET),
      shiftwindow(OLD),
      RET><1,!,
      change(source(TXT1)),
      refreshstatus.
pdwaction(2,0) :- refreshstatus.
pdwaction(3,1) :-
      readfilename(5,40,66,66,txt,"",NEW),
      change(filename(NEW)),
      file_str(NEW,NEWSOURCE),!,
        change(source(NEWSOURCE)),
        shiftwindow(OLDW),
        shiftwindow(1),
        window_str(NEWSOURCE),
        shiftwindow(OLDW),
        refreshstatus.
  pdwaction(3,1).
  pdwaction(3,2) :-
        source(SOURCE),
        filename(OLD),
        readfilename(5,40,66,66,txt,OLD,NEW),
        change(filename(NEW)),
        file_str(NEW,SOURCE),
        refreshstatus,!.
  pdwaction(3,2).
  pdwaction(3,3) :- setdir(5,40,66,66).
  pdwaction(4,0) :-
        lineinput(5,70,40,66,66,"Are you sure (y/n) ? ","",ANS),
        upper_lower(ANS,ANS1),
        ANS1><"y".

filename("minigol.txt").
source("").

goal
/*
          1         2         3         4         5         6         7
01234567890123456789012345678901234567890123456789012345678901234567890123456789
       COMPILE           EDIT              FILES                       QUIT
*/

  makewindow(1,23,23,"Edit",3,0,21,80),
  makestatus(112," Select with arrows or use first uppercase letter"),
  pulldown(66,[curtain(7,"Compile",[]),
               curtain(26,"Edit",[]),
               curtain(44,"Files",["Load","Save","Dir"]),
               curtain(64,"Quit"      ,[])
              ],_,_).
```

Using the XMINIGOL example files as a basis, you can easily implement the scanner (lexical analyzer) and parser for your own compiler. In order to experiment with your own source language, you can modify the example program this way:

1. Generate a new parser with the parser generator.
2. Change the scanner, XMINIGOL.SCA, so that it recognizes the new keywords.
3. Include the new scanner, parser, and domain definitions in a main program that should be a copy of XMINIGOL.PRO. Change the call of the top production name from *s_sent* to the name of the new top production in the newly generated parser.

The Parser Generator System

Before using the parser generator, it is necessary to compile it to an .EXE file. The parser generator is built from the following files, which are all on the distribution disks:

PARSER.GRM	– the grammar for parser input
PARSGEN.PRO	– the parser generator
PARSMAIN.PRO	– the parser generator's user interface
PARSER.SCA	– the scanner
PARSER.PAR	– the parser
PARSER.DEF	– global definitions
PARSER.PRJ	– project description

The parser generator also requires these previously defined files:

LINEINP.PRO
FILENAME.PRO
STATUS.PRO
PULLDOWN.PRO

To compile the parser-generation system, select Options from the Turbo Prolog main menu and choose Compile Project. Type in the project name PARSER.PRJ, then select Compile.

Like Turbo Prolog itself, the parser system uses a pull-down menu through which it is possible to load, edit, and save a grammar as a text file. The first step is to load the file containing the prepared parser-generator input (or to type it in directly using the Edit option). Activating the main-menu entry Generate Parser, now generates the required parser, which consists of two parts, each part in a separate file.

If the name of the file containing the parser-generator input is PASCAL.GRM, the generated parser's domain declarations will be in the file PASCAL.DOM, and the predicate declarations and clauses will be generated in the file PASCAL.PAR. The generated parser can be viewed by selecting Display Parser from the main menu.

The System Around the Generated Parser

The generated parser is, of course, a Turbo Prolog program. To detect syntax errors in source-language input, it contains just two predicates that should be adequate for most purposes. These two predicates—*expect* and *syntax_error*—have the following declarations:

```
predicates
   expect(CURSORTOK,TOKL,TOKL)
   syntax_error(STRING,TOKL)
```

expect is called when a given grammatical entity should be followed by a token. It is called with the expected token as the first parameter and the sublist of the scanned list currently being processed as the second parameter. The tail of this list is returned in the third parameter if its head is the correct token; otherwise, *expect* fails.

syntax_error is called when no matching production is found in a group of productions. It is called with the name of the production as its first parameter and the sublist of the scanned list currently being processed as its second parameter.

The first production tried is not always the correct one. Some of the *expect* and *syntax_error* calls that succeed may not correspond to a syntax error: A real syntax error occurs only if no matching production can be found.

Such a syntax error can be isolated by remembering the deepest level reached in the source text. Each time *expect* or *syntax_error* is called, the new cursor position is compared to the old. If the new cursor position is greater than the previous position, it should be saved as the deepest level achieved. Experience shows that this works pretty well but, as always, you should experiment with other methods.

Bootstrapping a Parser

Bootstrapping a parser means that you can take a parser generator and input a grammar that describes itself. The parser generator then generates another parser generator. In other words, it clones itself.

A common question is, Where did the first parser generator come from? The first one in this case is the Toolbox parser generator, which has been written for you. Now that you have it, however, you can do what is called "bootstrapping" and use it to generate the parser generator again.

All you need to do is treat the BNF grammar given in Table 7.1 as the grammar of the source language for which you want to generate a parser. You need to augment this grammar as done with the simple expressions in the previous section "Creating Your Own Grammar." The following grammar shows the result. It assumes the following tokenized forms for terminal symbols in the parser generator's grammar:

Terminal Symbol	**Tokenized Form**
->	*arrow*
=	*equal*
*	*star*
+	*plus*
userdefined	*userdefined_*
predicates	*predicates_*
domains	*domains_*
--	*priorsep*
,	*comma*
separator	*separator_*
:	*colon*
(	*lpar*
)	*rpar*

```
productions PARSER = SECTION*

SECTION =
   userdefined_predicates_ PRODNAMES -> userpreds_(PRODNAMES),
   userdefined_domains_ PRODNAMES -> userdoms_(PRODNAMES),
   productions_PRODUCTIONS -> productions_(PRODUCTIONS)

PRODUCTIONS = PRODUCTION+

PRODUCTION  = PRODNAME equal PRODBODY -> p(PRODNAME,PRODBODY)

PRODBODY = upper(STRING):CURSOR STAR_PLUS SEPARATOR ->
           list(PRODNAME,CURSOR,STAR_PLUS,SEPARATOR),
           PRODGROUPS -> groups(PRODGROUPS)

STAR_PLUS = star -> star,
           plus -> plus

SEPARATOR = separator_id(STRING) -> sep(STRING), -> none

PRODGROUPS = PRIORGROUP+ separator priorsep

PRIORGROUP = SINGPROD+ separator comma
```

```
SINGPROD = ASSOC GRAMTOKL arrow:CURSOR TERM ->
           prod(ASSOC,GRAMTOKL,CURSOR,TERM)

ASSOC = rightassoc_ -> right,
                    -> left

GRAMTOKL = GRAMTOK*

GRAMTOK = upper(STRING):CURSOR -> prodname(PRODNAME,CURSOR),
          TOKK CURSORDEMAND -> tok(TOKK,CURSORDEMAND)

CURSORDEMAND = colon PRODNAME -> curdemand(PRODNAME), -> none

TOKK = id(STRING):CURSOR lpar PRODNAMES rpar ->
          cmp(STRING,CURSOR,PRODNAMES),
       id(STRING):CURSOR -> name(STRING,CURSOR)

TERM = upper(STRING):CURSOR -> dom(PRODNAME,CURSOR),
       id(STRING):CURSOR lpar PRODNAMES rpar ->
          term(STRING,CURSOR,PRODNAMES),
       id(STRING):CURSOR -> name(STRING,CURSOR)

PRODNAME = upper(STRING) -> PRODNAME

PRODNAMES = PRODNAME+ separator comma
```

P A R T

2

Reference Guide

C H A P T E R

8

Reference Guide

Introduction

This section defines and describes, in alphabetical order, all the Turbo Prolog Toolbox tools. All the essential information about each tool—predicate, declaration, flow pattern, function, parameters, environment, and so on—is included. The environment comprises the programs required to run the tool.

In addition, the following table summarizes the environments of all the Toolbox files, listing the files required by each specific file.

The files are listed in alphabetical order, under the headings Prolog Files, Object Files, and Example Files.

Table 8.1: The Turbo Prolog Toolbox Files

Prolog Files	
BIOS	None
BOXMENU	TDOMS.PRO, TPREDS.PRO
COMGLOBS	None
DBASE3	TDOMS.PRO, TPREDS.PRO, READEXT.PRO, REALINTS.OBJ
FILENAME	TDOMS.PRO, TPREDS.PRO, LINEINP.PRO
GBAR	TDOMS.PRO, GDOMS.PRO, GGLOBS.PRO, TPREDS.PRO, GPREDS.PRO, GRAPHICS.OBJ, EGAGRAPH.OBJ
GDOMS	None
GEGA	TDOMS.PRO
GGLOBS	GDOMS.PRO
GGRAPH	TDOMS.PRO, GDOMS.PRO, TPREDS.PRO, GPREDS.PRO
GPIE	TDOMS.PRO, TPREDS.PRO, GDOMS.PRO, GPREDS.PRO, GRAPHICS.OBJ EGAGRAPH.OBJ
GPREDS	GDOMS.PRO
HELP	TDOMS.PRO, TPREDS.PRO
HELPDEF	TDOMS.PRO, LONGMENU.PRO, STATUS.PRO, TPREDS.PRO,LINEINP.PRO, FILENAME.PRO, SCRHND.PRO, HELP.SCR
HNDBASIS	TDOMS.PRO, TPREDS.PRO, MENU.PRO, STATUS.PRO, LINEINP.PRO, SCRHND.PRO or VSCRHND.PRO
LINEINP	TDOMS.PRO, TPREDS.PRO
LINEMENU	TDOMS.PRO, TPREDS.PRO
LONGMENU	TDOMS.PRO, TPREDS.PRO
LOTUS	TDOMS.PRO, TPREDS.PRO, READEXT.PRO, REALINTS.OBJ
MENU	TDOMS.PRO, TPREDS.PRO
MIXSCR	GOODS.SCR, ORDER.SCR, CUSTOMER.SCR, INF.SCR, INVOICE.SCR
PARSGEN	PARSER.PRJ, PARSER.DEF, TPREDS.PRO
PARSMAIN	PARSER.PRJ (PARSGEN.OBJ), PARSER.DEF, TPREDS.PRO, LINEINP.PRO, FILENAME.PRO, STATUS.PRO, PULLDOWN.PRO, PARSER.SCA, PARSER.PAR
PULLDOWN	TDOMS.PRO, TPREDS.PRO
READEXT	None
REFLEX	TDOMS.PRO, TPREDS.PRO, READEXT.PRO, REALINTS.OBJ
REPORT	TDOMS.PRO
RESIZE	STATUS.PRO, TDOMS.PRO, TPREDS.PRO
SCRDEF	TDOMS.PRO, TPREDS.PRO, MENU.PRO, STATUS.PRO, LINEINP.PRO, FILENAME.PRO, RESIZE.PRO

SCRHND	TDOMS.PRO, TPREDS.PRO, LINEINP.PRO, STATUS.PRO
STATUS	TDOMS.PRO, TPREDS.PRO
TDOMS	None
TPREDS	TDOMS.PRO
TREE	TDOMS.PRO, TPREDS.PRO
VSCRHND	TDOM.PRO, TPREDS.PRO, LINEINP.PRO, STATUS.PRO

Object Files

SERIAL.OBJ	COMGLOBS.PRO
GRAPHICS.OBJ	GDOMS.PRO, GGLOBS.PRO
PICTOOLS.OBJ	GDOMS.PRO, GGLOBS.PRO
TICKS.OBJ	COMGLOBS.PRO
EGAGRAPH.OBJ	

Example Files

XBAR	XBAR.PRJ, TDOMS.PRO, GDOMS.PRO, GLOBS.PRO, TPREDS.PRO, GPREDS.PRO, GBAR.PRO, GRAPHICS.OBJ, EGAGRAPH.OBJ
XBOXMENU	TDOMS.PRO, TPREDS.PRO, BOXMENU.PRO, STATUS.PRO
XCAREER	TDOMS.PRO, TPREDS.PRO, MENU.PRO
XCLUB	TDOMS.PRO, TPREDS.PRO, MENU.PRO, STATUS.PRO, LINEINP.PRO, FILENAME.PRO, SCRHND.PRO, XCLUB.SCR, XCLUB.DBA
XCOMMU	XCOMMU.PRJ (SERIAL.OBJ, MODEM.OBJ, TRICKS.OBJ), TDOMS.PRO, COMGLOBS.PRO, TPREDS.PRO, MENU.PRO
XDBASE3	XDBASE3.PRJ, READEXT.PRO, DBASE3.PRO, REALINTS.OBJ, XDBASE3.DBF, XDBASE3.DBT
XEGA	TDOMS.PRO, GDOMS.PRO, TPREDS.PRO, GPREDS.PRO, GEGA.PRO
XFILENAM	TDOMS.PRO, TPREDS.PRO, LINEINP.PRO, FILENAME.PRO, STATUS.PRO
XGEOMETR	XGEOMETR.PRJ (GRAPHICS.OBJ), EGAGRAPH.OBJ, TDOMS.PRO, GDOMS.PRO, GGLOBS.PRO, TPREDS.PRO, GPREDS.PRO
XGRAPH	TDOMS.PRO, GDOMS.PRO, GGLOBS.PRO, TPREDS.PRO, GPREDS.PRO, GGRAPH.PRO, GRAPHICS.OBJ, EGAGRAPH.OBJ
XHELP	TDOMS.PRO, HELP.PRO
XIQ	TDOMS.PRO, TPREDS.PRO, MENU.PRO
XLABEL	TDOMS.PRO, STATUS.PRO, TPREDS.PRO, MENU.PRO, LINEINP.PRO, FILENAME.PRO, SCRHND.PRO, XLABEL.DBA

XLCAREER	TDOMS.PRO, TPREDS.PRO, MENU.PRO
XLINEINP	TDOMS.PRO, TPREDS.PRO, LINEINP.PRO, STATUS.PRO
XLINEMNU	TDOMS.PRO, TPREDS.PRO, LINEMENU.PRO, STATUS.PRO
XLONGMNU	TDOMS.PRO, TPREDS.PRO, LONGMENU.PRO, STATUS.PRO
XLOTUS	LOTUS.PRJ, (REALINTS.OBJ) , READEXT.PRO, LOTUS.PRO, XLOTUS.WRK
XMADDER	TDOMS.PRO, TPREDS.PRO, MENU.PRO, STATUS.PRO, LINEINP.PRO, SCRHND.PRO, XMADDER.SCR
XMAIL	HNDBASIS.PRO, XMAIL.SCR
XMENU	TDOMS.PRO, TPREDS.PRO, STATUS.PRO, MENU.PRO
XMINIGOL	TDOMS.PRO, TPREDS.PRO, STATUS.PRO, PULLDOWN.PRO, LINEINP.PRO, FILENAME.PRO, XMINIGOL.DOM, XMINIGOL.SCA, XMINIGOL.PAR
XPARS.DOM	None
XPARS.GRM	None
XPARS.PAR	XPARS.DOM
XPARS.PRO	XPARS.DOM, XPARS.SCA, XPARS.PAR
XPARS.SCA	XPARS.DOM
XPICDEMO	XPICDEMO.PRJ (PICTOOLS.OBJ), TDOMS.PRO, GDOMS.PRO, GGLOBS.PRO, TPREDS.PRO, WELCOME.PIC, TEST.PIC
XPIE	XPIE.PRJ (GRAPHICS.OBJ, EGAGRAPH.OBJ), TDOMS.PRO, GDOMS.PRO, GGLOBS.PRO, TPREDS.PRO, GPREDS.PRO, GPIE.PRO
XPOLLING	XPOLLING.PRJ (SERIAL.OBJ, TICKS.OBJ), TDOMS.PRO, COMGLOBS.PRO, TPREDS.PRO, MENU.PRO
XPRINTER	XPRINTER.PRJ, (SERIAL.OBJ, TICKS.OBJ), COMGLOBS.PRO
XPULLDW	TDOMS.PRO, TPREDS.PRO, STATUS.PRO, PULLDOWN.PRO
XREFLEX	XREFLEX.PRJ (REALINTS.OBJ), READEXT.PRO, REFLEX.PRO, REFLEX.RXD
XREPORT	TDOMS.PRO, TPREDS.PRO, REPORT.PRO, XREPORT.SCR
XRESIZE	TDOMS.PRO, TPREDS.PRO, STATUS.PRO, RESIZE.PRO
XSHOP	TDOMS.PRO, XSHOP.SCR, TPREDS.PRO, MENU.PRO, LINEINP.PRO, STATUS.PRO, SCRHND.PRO
XSLIDES	XSLIDES.PRJ (PICTOOLS.OBJ), TDOMS.PRO, TPREDS.PRO, MENU.PRO
XSTATUS	TDOMS.PRO, TPREDS.PRO, STATUS.PRO
XTERM	XTERM.PRJ, (SERIAL.OBJ, TICKS.OBJ), TDOMS.PRO, COMGLOBS.PRO, TPREDS.PRO, STATUS.PRO
XTREE	TDOMS.PRO, TPREDS.PRO, STATUS.PRO, MENU.PRO, TREE.PRO

XVIRTUAL	TDOMS.PRO, TPREDS.PRO, MENU.PRO, STATUS.PRO, LINEINP.PRO, VSCRHND.PRO, XVIRTUAL.SCR
XXMODEM	XXMODEM.PRJ (SERIAL.OBJ, TICKS.OBJ), TDOMS.PRO, COMGLOBS.PRO, TPREDS.PRO, MENU.PRO

axislabels

Predicate
axisLabels(AxesNo,XaxisText,YaxisText)

Declaration
axisLabels(Integer,String,String)

Flow pattern
axisLabels(i,i,i)

Function
Used to label a pair of axes.

Parameters
AxesNo identifies the pair of axes to be affected and is the axes identifier returned from *makeAxes*. *XaxisText* and *YaxisText* are the strings to be used for labeling the X and Y axes respectively.

Remarks
In order to use *axisLabels,* the following **database** predicates must be declared in the containing program:

```
database
   Scale(ScaleNo,x,x,y,y)
   activeScale(ScaleNo)
   axes(Integer,Integer,Integer,Xmarker,Ymarker,Col,Row,Col,Row)
```

Example
axisLabels(1,"Length","Height")

Contained in
GGRAPH.PRO

Environment
TDOMS.PRO, GDOMS.PRO, TPREDS.PRO, GPREDS.PRO

See also
modifyAxes, makeAxes, refreshAxes

bargraph

Predicate
barGraph(Left,Bottom,Right,Top,BarRatio,BarList,Factor)

Declaration
barGraph(Col,Row,Col,Row,BarRatio,BarList,Factor)

Flow pattern
barGraph(i,i,i,i,i,i,o)

Function
The predicate *barGraph* makes it possible to display numerical data in bar-chart form.

Parameters
The first four parameters determine how much space there should be between the borders of the active window in which the bar chart is drawn and the edge of the screen. The parameter *BarRatio* specifies the width of the bars relative to the spacing between the bars. A spacing of 0.5 indicates that the bars have the same width as the gaps between them.

BarList belongs to the domain

```
BARLIST = BAR*
BAR     = bar(VHEIGHT,STRING,COLOR,COLOR) ; space
```

Normally a *bar* is specified by its height, a label (which may be an empty string), a frame color, and a fill color. A bar also can be a *space* that separates the bars visually. The bars are automatically scaled to fill the specified area of the active window. The scaling factor is returned as the last parameter.

Example
```
barGraph(3,4,4,4,0,5,
        [bar(2,"",1,2),
         bar(3,"1984",1,3),
         bar(5,"",1,2),
         space,
         bar(4,"",1,2),
         bar(7,"1985",1,3),
         bar(7,"",1,2)],_)
```

Contained in
GBAR.PRO

Environment
TDOMS.PRO, GDOMS.PRO, GGLOBS.PRO, TPREDS.PRO, GPREDS.PRO

See also
barGraph3d

barGraph3d

Predicate
barGraph3d(Left,Bottom,Right,Top,BarRatio,Angle,BarList,Factor)

Declaration
barGraph3d(Col,Row,Col,Row,BarRatio,Theta,BarList,Factor)

Flow pattern
barGraph3d(i,i,i,i,i,i,i,o)

Function
The predicate *barGraph3d* lets you display numerical data in bar-chart form in which the bars appear three-dimensional.

Parameters
The first four parameters are text coordinates, used to determine how much space there should be between the borders of the active window in which the bar chart is drawn and the edge of the screen.

The parameter *Angle* specifies the angle from which the three-dimensional bars are to be viewed, measured in radians.

The parameter *BarRatio* specifies how much space the bars take up relative to the spacing between the bars. A spacing of 0.5 indicates equal space for the bars and the gaps between them.

BarList belongs to the domain

```
BARLIST = BAR*
BAR     = bar(VHEIGHT,STRING,COLOR,COLOR) ; space
```

Normally a bar is specified by its height, a label (which may be an empty string), a frame color, and a fill color. A bar can also be a space to separate the bars visually. The bars are automatically scaled to fill the specified area of the active window. The scaling factor is returned as the last parameter.

Example
barGraph3d(3,4,4,4,0.55,0.5,[bar(2,"",1,2),space,bar(3,"1984",1,3)],_),

Contained in
GBAR.PRO

Environment
TDOMS.PRO, GDOMS.PRO, GGLOBS.PRO, TPREDS.PRO, GPREDS.PRO

See also
barGraph

border

Predicate
border(Color)

Declaration
border(Integer)

Flow pattern
border(i)

Function
Sets the screen border color in text mode using a CGA graphics card. (It is also possible to set the border color using an EGA graphics card. Refer to the tool predicate *setEGAregister*.

Parameters
Color specifies the desired screen border color as one of the 16 possibilities, coded as shown in Table 8-3 of the *Turbo Prolog Owner's Handbook.*

Example
border(15)

Contained in
BIOS.PRO

Environment
TDOMS.PRO, TPREDS.PRO

See also
setEGAregister

box

Predicate
box(Row,Col,Row2,Col2,LineColor,FillColor,Fill)

Declaration
box(VRow,VCol,VRow,VCol,Color,Color,Fill) – (i,i,i,i,i,i,i) language c

Flow pattern
box(i,i,i,i,i,i,i)

Function
Draws a four-sided box shape on the graphics display screen.

Parameters
box draws a box using virtual coordinates with the upper left-hand corner at *(Row,Col)* and lower right-hand corner *(Row2,Col2)* with the indicated *Color*. Depending on the value of *Fill*, the box is filled with the color *FillColor*. The *Fill* parameter can be 1 or 0, indicating that the box will or will not be filled, respectively.

Remarks
The predicate *box* is implemented in C.

Range for *Rows*: 0-31999
Range for *Columns*: 0-31999

It requires the creation of a project.

Example
box(2000,2000,20000,20000,1,2,0)

Contained in
GRAPHICS.OBJ

Environment
GDOMS.PRO, GGLOBS.PRO

boxmenu

Predicate
boxmenu(PosRow,PosCol,NoOfRows,NoOfCols,Wattr,Fattr,ItemList,Title,InitItem,Choice)

Declaration
boxmenu(Row,Col,Row,Col,Attr,Attr,StringList,String,Integer,Integer)

Flow pattern
boxmenu(i,i,i,i,i,i,i,i,i,o)

Function
The *boxmenu* predicate allows a menu selection to be made from a collection of menu items displayed across a row.

Parameters
PosRow and *PosCol* define the upper left-hand corner of the window that contains the menu. The size of the window is determined by *NoOfRows* and *NoOfCols*. *Wattr* and *Fattr* specify the attributes of the window and its frame, respectively. *ItemList* is a list of strings, one for each menu item. *Title* determines the text in the frame of the window. When the menu is displayed, menu choice number *InitItem* is highlighted by the cursor. *Choice* is an integer indicating the selection made, coded as

0 = Esc was pressed
1 = First menu item selected
2 = Second menu item selected

and so on.

Remarks
The window containing the menu is automatically adjusted if it is placed too far to the left or too near to the bottom of the screen. The predicate fails if the list of menu items has length zero.

Example
boxmenu(5,10,5,50,7,7,[Basic,Pascal,Lisp,Prolog],"Language",2,CHOICE)

Contained in
BOXMENU.PRO

Environment
TDOMS.PRO, TPREDS.PRO

Recognized keys
Cursor keys, Esc, F10, Return, Home, End, PgUp, PgDn

See also
boxmenu_leave, boxmenu_mult

boxmenu_leave

Predicate
boxmenu_leave(PosRow,PosCol,NoOfRows,NoOfCols,Wattr,Fattr,ItemList,Title,InitItem,Choice)

Declaration
boxmenu_leave(Row,Col,Row,Col,Attr,Attr,StringList,String,Integer,Integer)

Flow pattern
boxmenu_leave(i,i,i,i,i,i,i,i,i,o)

Function
The *boxmenu_leave* predicate allows menu selection from more than one column. After selection, the menu is left on the screen.

Parameters
PosRow and *PosCol* define the upper left-hand corner of the window that contains the menu. *NoOfRows* and *NoOfCols* determine the size of the window. *Wattr* and *Fattr* specify the attributes of the window and its frame, respectively. *ItemList* is a list of strings, one for each menu item. *Title* determines the text in the frame of the window. When the menu is displayed, menu choice number *InitItem* is highlighted by the cursor. *Choice* is an integer indicating the selection made coded as

0 = Esc was pressed
1 = First menu item selected
2 = Second menu item selected

and so on.

Remarks
The window containing the menu is automatically adjusted if it is placed too far to the left or too near to the bottom of the screen. The predicate fails if the list of menu items has length zero.

Example
boxmenu_leave(5,10,5,50,7,7,[Basic,Pascal,Lisp,Prolog],"Language",2,CHOICE)

Contained in
BOXMENU.PRO

Environment
TDOMS.PRO, TPREDS.PRO

Recognized keys
Cursor keys, Esc, F10, Return, Home, End, PgUp, PgDn

See also
boxmenu, boxmenu_mult

boxmenu_mult

Predicate
boxmenu_mult(Row,Col,NoOfRows,NoOfCols,Wattr,Fattr,ItemList,Title,InitItems,ChoiceList)

Declaration
boxmenu_mult(Row,Col,Row,Col,Attr,Attr,StringList,String,IntegerList,IntegerList)

Flow pattern
boxmenu_mult(i,i,i,i,i,i,i,i,i,o)

Function
The *boxmenu_mult* predicate works like *boxmenu* but allows more than one selection to be made.

Parameters
Row and *Col* define the position of the upper left-hand corner of the window. *NoOfRows* and *NoOfCols* determine the size of the window containing the menu. *Wattr* and *Fattr* specify the attributes of the window and its frame, respectively. *ItemList* is a list of strings, one for each menu item. *Title* determines the text in the frame of the window. When the menu is displayed, menu choice numbers *InitItems* is highlighted by the cursor. *ChoiceList* is an integer list that becomes instantiated to the integer codes for the selections made. If Esc is pressed during menu selection, *ChoiceList* is bound to an empty list.

Remarks
The window containing the menu is automatically adjusted if it is placed too far to the left or too near the bottom of the screen. The predicate fails if the list of menu items has length zero.

Example
boxmenu_mult(5,10,5,50,7,7,[Basic,Pascal,Lisp,Prolog],"Language",[3],CHOICES)

Contained in
BOXMENU.PRO

Environment
TDOMS.PRO, TPREDS.PRO

Recognized keys
Cursor keys, Esc, F10, Return, Home, End, PgUp, PgDn

See also
boxmenu, boxmenu_leave

changestatus

Predicate
changestatus(StringEntry)

Declaration
changestatus(String)

Flow pattern
changestatus(i)

Function
Used to change the text in the status window.

Parameters
The text to which *StringEntry* is bound is displayed in the status line.

Example
changestatus("Press H for HELP")

Contained in
STATUS.PRO

Environment
TDOMS.PRO, TPREDS.PRO

See also
makestatus, refreshstatus, removestatus, tempstatus

closeRS232

Predicate
closeRS232(PortNo)

Declaration
closeRS232(Integer) – (i) language c

Flow pattern
closeRS232(i)

Function
The tool predicate *CloseRS232* closes an open communication port so that the PC interrupt mechanisms are restored to the state before the corresponding *OpenRS232* was executed and the input/output buffers de-allocated. It is extremely important to close a communication port before an application terminates because the interrupt routines redirect interrupts IRQ3 and IRQ4 from the interrupt controller. *CloseRS232* fails if the communication port referred to is not open or does not exist.

Parameters
If *PortNo* is bound to 1, COM1 is closed; if *PortNo* is bound to 2, COM2 is closed.

Remarks
Requires the creation of a project.

Example
closeRS232(1)

Contained in
SERIAL.OBJ

Environment
COMGLOBS.PRO

See also
openRS232

createwindow (used by *scrhnd*)

Predicate
createwindow(TopLineSwitch)

Declaration
createwindow(Symbol)

Flow pattern
createwindow(i)

Function
Creates a window through which the display created by *scrhnd* may be viewed.

Parameters
If *TopLineSwitch* is bound to ON, then space is reserved in the window for a line that gives the name of the field containing the cursor. If *TopLineSwitch* is bound to OFF, no such reservation is made.

Remarks
It's not really a tool on its own but should be regarded as being closely related to *scrhnd*.

Example
createwindow(on)

Contained in
SCRHND.PRO

Environment
TDOMS.PRO, TPREDS.PRO, LINEINP.PRO, STATUS.PRO

See also
scrhnd

defineScale

Predicate
defineScale(ScaleNo,X0,X1,Y0,Y1)

Declaration
defineScale(ScaleNo,X,X,Y,Y)

Flow pattern
defineScale(i,i,i,i,i)

Function
Defines a scale for a set of axes.

Parameters
(X0,X1) and (Y0,Y1) are the scaled limits of the X and Y axes.

Example
defineScale(1,0,100,0,100)

Contained in
GGRAPH.PRO

Environment
TDOMS.PRO, GDOMS.PRO, TPREDS.PRO, GPREDS.PRO

delInBuf_RS232

Predicate
delInBuf_RS232(PortNo)

Declaration
delInBuf_RS232(Integer) – (i) language c

Flow pattern
delInBuf_RS232(i)

Function
Deletes the input queue.

Parameters
With *PortNo* bound to the code for an I/O port, it deletes the contents of the input buffer for that port.

Remarks
It can be useful in cases where data input during a transmission phase should be suppressed. *delInBuf_RS232* fails if the specified port is not open.

Requires the definition of a project.

Example
delOutBuf(1)

Contained in
SERIAL.OBJ

Environment
COMGLOBS.PRO

See also
delOutBuf_RS232

delOutBuf_RS232

Predicate
delOutbuf_RS232(PortNo)

Declaration
delOutBuf_RS232(Integer) – (i) language c

Flow pattern
delOutBuf_RS232(i)

Function
Deletes the output queue.

Parameters
With *PortNo* bound to the code for an I/O port, it deletes the contents of the output buffer for that port.

Remarks
This predicate can be useful when it is necessary to retransmit a certain block of data. It fails if the specified port is not open.

Example
delOutBuf_RS232(2)

Contained in
SERIAL.OBJ

Environment
COMGLOBS.PRO, TICKS.OBJ

See also
delInBuf_RS232

diskspace

Predicate
diskspace(Disknumber,TotalNoOfBytes,NoOfFreeBytes)

Declaration
diskspace(Integer,Real,Real)

Flow pattern
diskspace(i,o,o)

Function
Returns the total number of bytes available and the number of free bytes on the disk in the selected drive.

Parameters

Disknumber	= 0	Currently active drive
Disknumber	= 1	Drive A:
Disknumber	= 2	Drive B:
Disknumber	= 3	Drive C:

Remarks
Also described in the *Turbo Prolog Owner's Handbook* (see "Low-Level Support" in Chapter 11).

Example
diskspace(0,Total,Free)

Contained in
BIOS.PRO

dosver

Predicate
dosver(DosVersionNumber)

Declaration
dosver(Real)

Flow pattern
dosver(o)

Function
Returns the version number of the DOS system in use.

Parameters
DosVersionNumber is bound to a real that corresponds to the DOS version number.

Remarks
Also described in the *Turbo Prolog Owner's Handbook* (see "Low-Level Support" in Chapter 11).

Example
dosver(OurDOSNo)

Contained in
BIOS.PRO

draw

Predicate
draw(MyDrawings)

Declaration
draw(Drawings)

Flow pattern
draw(i)

Function
The tool predicate *draw* is used to draw a scaled polygon from the domain DRAWINGS.

Parameters
Draws the object *Drawings* from the domain DRAWINGS. This is a list of colored polygons in which the functor *d* binds a color to a DRAWING, which is itself a list of points that should be connected to a polygon:

```
X,Y = REAL
POINT = p(X,Y)
DRAWING = POINT*
D = d(ColOR,DRAWING)
DRAWINGS = D*
```

Remarks
The drawing speed can be increased by declaring the domains *X* and *Y* as INTEGERs instead of REALs.

Example
draw([d(1,[p(70,100),p(90,195)]),d(3,[p(400,300),p(200,205)])])

Contained in
GGRAPH.PRO

Environment
TDOMS.PRO, GDOMS.PRO, TPREDS.PRO, GPREDS.PRO,

ellipse

Predicate
ellipse(Row,Col,Radius1,Ratio,Color,FillColor,Fill)

Declaration
ellipse(VRow,VCol,Radius,Real,Color,Color,Fill) – (i,i,i,i,i,i,i) language c

Flow pattern
ellipse(i,i,i,i,i,i,i)

Function
Draws an ellipse in the currently active graphics window.

Parameters
The ellipse has its center at (*Row,Col*) and is drawn using the indicated *Color*. The radius along the horizontal axis is *Radius1*, and the radius along the vertical axis is scaled according to the *Ratio* parameter. *Row, Col,* and *Radius* are all given as virtual coordinates. If *Fill* is bound to 1, the ellipse is filled out with the color *FillColor*.

Remarks
The predicate *ellipse* is coded in C.

Example
ellipse(5000,5000,4000,0.5,1,2,1)

Contained in
GRAPHICS.OBJ

Environment
GDOMS.PRO, GGLOBS.PRO

field_action (programmer defined)

Predicate
field_action(MyFieldName)

Declaration
field_action(FName)

Flow pattern
field_action(i)

Function
Called by *scrhnd* in conjunction with a screen definition created using SCRDEF.PRO to determine the actions to be taken when Return is pressed in a given field.

Parameters
Defines the action(s) to be taken when Return is pressed in the field to which *MyFieldName* is bound.

Remarks
Default defining clauses for *field_action* are given in HNDBASIS.PRO (see "Associating Actions with Fields" in Chapter 3). Other defining clauses must be added by the programmer.

Example
field_action(dir) :- setdir(......)

Contained in
SCRHND, VSCRHND

Environment
TPREDS.PRO, TDOMS.PRO, LINEINP.PRO, STATUS.PRO

See also
scrhnd

field_value (programmer defined)

Predicate
field_value(MyFieldName,StringForField)

Declaration
field_value(FName,String)

Flow pattern
field_value(i,o)

Function
Called by *scrhnd* in conjunction with a screen definition, created using SCRDEF.PRO, to determine the value of a given field.

Parameters
Defines the value of the field to which *MyFieldName* is bound.

Remarks
Default defining clauses for *field_value* are given in HNDBASIS.PRO (see "Associating Values with Fields" in Chapter 3). Other defining clauses must be added by the programmer.

Example
field_value(disk,DISK) :- disk(DISK).

Contained in
SCRHND, VSCRHND

Environment
TPREDS.PRO, TDOMS.PRO, LINEINP.PRO, STATUS.PRO

See also
scrhnd

Findmatch

Predicate
findmatch(FileSpec,Attr,FileName,FileAttr,Hour,Min,Year,Month,Day,Size)

Declaration
findmatch(String,Integer,String,Integer,Integer,Integer,Real,Integer,Integer,Integer)

Flow pattern
findmatch(i,i,o,o,o,o,o,o,o,o)

Function
The predicate *findmatch* searches the DOS file directory for file names that match a given string mask and file attribute. For the matching files, all available information is returned.

Parameters
Filespec = The usual DOS filespec (for example, C:*.*)

Attr represents a bit-mask value determined according to the following table:

0	Search for ordinary files
1	File is read only
2	Hidden file
4	System file
8	Volume label
16	Subdirectory
32	Archive file (used by backup & restore)

Hour is in the range 0 to 23, denoting the hour of the day when the file was created; *Min* is in the range 0 to 59, representing the minutes past the hour when the file was created; and *Year*, *Month*, and *Day* are the appropriate values in the ranges 1980 to 2099, 1 to 12, and 1 to 31, respectively. *FilesSize* denotes the size of the file (and is in the range 0 to 30 MB).

Example
findmatch(".*",63,Name,Attrib,Hour,Min,Year,Month,Day,Size)*

Contained in
BIOS.PRO

findScale

Predicate
findScale(ScaleNo,MyDrawing,Xfactor,Yfactor)

Declaration
findScale(ScaleNo,Drawing,Factor,Factor)

Flow pattern
findScale(o,i,i,i)

Function
Evaluates a graphics image and then automatically defines an appropriate scale so that the image can fit into a given window.

Parameters
The image is represented as a list of points, *MyDrawing,* belonging to the domain DRAWING, which is defined as

```
X,Y = REAL
POINT = p(X,Y)
DRAWING = POINT*
```

Xfactor indicates the amount of stretch (if any) desired in the X direction, when the images specified for this scale are represented on screen; *Yfactor* indicates the amount of stretch in the Y direction expressed as a ratio to the original.

Example
findScale(X,[p(1,1),p(9,1),p(1,9)],2,1.3)

Contained in
GGRAPH.PRO

Environment
TDOMS.PRO, GDOMS.PRO, TPREDS.PRO, GPREDS.PRO, GGRAPH.PRO

See also
defineScale, shiftScale

getverify

Predicate
getverify(Switch)

Declaration
getverify(Integer)

Flow pattern
getverify(o)

Function
getverify returns the state of the DOS verify switch. When the verify switch is ON, every write operation in the DOS system will be followed by a read operation to ensure that the data has been stored correctly.

Parameters
If *Switch* is bound to 0, then the DOS verify switch is OFF; if *Switch* is bound to 1, then the DOS verify switch is ON.

Example
getverify(X)

Contained in
BIOS.PRO

See also
setverify

gwrite

Predicate
gwrite(Row,Column,StringParam,Color,VerticalOrHorizontal)

Declaration
gwrite(Row,Col,String,Color,Integer)

Flow pattern
gwrite(i,i,i,i,i)

Function
Writes a string in a graphics window.

Parameters
The string *StringParam* is written on the screen starting at the point specified by (*Row,Col*) (regarded as text coordinates) in the given *Color*. If the parameter *VerticalOrHorizontal* is bound to 1, the string is displayed vertically downwards; if it is bound to zero, the string is displayed horizontally as is the normal way.

Example
gwrite(5,7,"SALES LAST MONTH",3,0)

Contained in
GPREDS.PRO

Environment
GDOMS.PRO, TDOMS.PRO

help

Predicate
help

Declaration
help

Flow pattern
None

Function
Displays the Help messages associated with the current Help context, determined by *push_helpcontext* and *pop_helpcontext*. These tool predicates use Help texts from the database predicate *helpcontext*.

Remarks
The text to be displayed should be created by using the utility program HELPDEF.PRO, which is described in "Context-Sensitive Help" in Chapter 2.

Example
help

Contained in
HELP.PRO

Environment
TDOMS.PRO, TPREDS.PRO

See also
help, helptext, pop_helpcontext, push_helpcontext, temp_helpcontext

init_dBase3

Predicate
init_dBase3(TotRecs,FldNameL,FldDescL)

Declaration
init_dBase3(Real,FldNameL,FldDescL)

Flow pattern
init_dBase3(o,o,o)

Function
The first step in accessing a dBASE III file from a Turbo Prolog program is to call the tool predicate *init_dBase3,* which builds data structures describing dBASE III records.

Parameters
```
FLDDESCL = FLDDESC*                                  /* Description for each field */
FLDDESC = FLDDESC(DBASE3TYPE,Integer)
DBASE3TYPE = ch;r;l;m;d

FLDNAMEL = String*
```

Remarks
Requires the definition of a project.

Contained in
DBASE3.PRO

Environment
TDOMS.PRO, TPREDS.PRO, READEXT.PRO, REALINTS.OBJ

See also
rd_dBase3File, rddBase3Rec

init_Reflex

Predicate
init_Reflex(TotRecs,FldNames,ReflexTypeL,TextPools)

Declaration
init_Reflex(Integer,FldNames,ReflexTypeL,TxtPools)

Flow pattern
init_Reflex(o,o,o,o)

Function
To access a Reflex file, it is first necessary to call the tool predicate *init_Reflex*, which builds Prolog data structures describing the Reflex data records.

Parameters
Totrecs is bound to the total number of records in the file. The other parameter is bound to structures from the following domains.

```
FLDNAMES = String*                                   /* Reflex field names */

REFLEXTYPEL = REFLEXTYPE*
REFLEXTYPE = u;t;rt;d;r;i                 /* Internal Reflex type for each field */

TXTPOOLS = TXTPOOL*
TXTPOOL = REPTXT*
REPTEXT = text(Integer,String)                             /* Indexed strings */
```

Contained in
REFLEX.PRO

Environment
TDOMS.PRO, TPREDS.PRO, READEXT.PRO, REALINTS.OBJ

See also
rd_ReflexFile, rd_ReflexRec

lineinput

Predicate
lineinput(Row,Col,Len,Wattr,Fattr,Prompt,BeforeString,AfterString)

Declaration
lineinput(Row,Col,Len,Attr,Attr,String,String,String)

Flow pattern
lineinput(i,i,i,i,i,i,i,o)

Function
Highlights a screen field by drawing a window around it, then accepts input from the user for that field. Once the user's input has been accepted, the window containing the field (and its contents) is deleted.

Parameters
Creates a window containing *BeforeString* at position (*Row,Col*) on the screen with attribute *Attr*, and allows input in the field of length *Len* characters indicated—either by typing new text or editing *BeforeString* using the arrow and delete keys. Once input is complete, the user types F10 or Return, and the modified string is returned in *AfterString*. If Esc is pressed during input, *lineinput* fails.

Remarks
In order to use *lineinput* the following **database** predicates must be declared in the containing program:

```
database
   insmode
   lineinpstate(String,Col)
   lineinpflag
```

If, during a call, the first key pressed is not a cursor key, then *BeforeString* is deleted. F8 can be used to re-establish the default text, while Ctrl-Backspace-Del deletes it.

The input text may be up to 64K—the text will scroll in the window if necessary.

Example
lineinput(3,5,40,7,7,"Address Label","Please type your first name",X)

Contained in
LINEINP.PRO

Environment
TDOMS.PRO, TPREDS.PRO

Recognized keys
Left and right cursor keys; Ctrl-Left arrow and Ctrl-Right arrow; Esc; F8; F10; Return; Home; End; Del; Backspace-Del; Ctrl-Backspace-Del; Ins

See also
lineinput_leave, lineinput_repeat

lineinput_leave

Predicate
lineinput_leave(Row,Col,Len,Wattr,Fattr,Prompt,BeforeString,AfterString)

Declaration
ineinput_leave(Row,Col,Len,Attr,Attr,String,String,String)

Flow pattern
lineinput_leave(i,i,i,i,i,i,i,o)

Function
Highlights a screen field by drawing a window around it, then accepts input from the user for that field. Once the user's input has been accepted, the window containing the field (and its contents) remains on the screen.

Parameters
Creates a window containing *BeforeString* at position (*Row,Col*) on the screen with attribute *Wattr*, and allows input in the field of length *Len* characters indicated—either by typing new text or editing *BeforeString* using the arrow and delete keys. Once input is complete, the user types F10 or Return, and the modified string is returned in *AfterString*. If Esc is pressed during input, *lineinput* fails.

Remarks
In order to use *lineinput_leave,* the following **database** predicates must be declared in the containing program:

```
database
   insmode
   lineinpstate(String,Col)
   lineinpflag
```

Example
lineinput_leave(3,5,40,7,7,"","Which printer font?",X)

Contained in
LINEINP.PRO

Environment
TDOMS.PRO, TPREDS.PRO

See also
lineinput

lineinput_repeat

Predicate
lineinput_repeat(Row,Col,Len,Wattr,Fattr,Prompt,BeforeString,AfterString)

Declaration
lineinput_repeat(Row,Col,Len,Attr,Attr,String,String,String)

Flow pattern
lineinput_repeat(i,i,i,i,i,i,o)

Function
Accepts input from a given screen field as with *lineinput,* but *lineinput_repeat* succeeds after each text input and stacks a backtracking point (unless Esc is pressed, in which case it fails).

Parameters
Creates a window containing *BeforeString* at position (*Row,Col*) on the screen with attribute *Wattr,* and allows input in the field of length *Len* characters indicated—either by typing new text or editing *BeforeString* using the arrow and Del keys. Once input is complete, the user types F10 or Return, and the modified string is returned in *AfterString*. If Esc is pressed during input, *lineinput* fails.

Remarks
In order to use *lineinput_repeat* the following **database** predicates must be declared in the containing program:

```
database
   insmode
   lineinpstate(String,Col)
   lineinpflag
```

Example
lineinput_repeat(3,4,40,7,7,"","Date:",Z)

Contained in
LINEINP.PRO

Environment
TDOMS.PRO, TPREDS.PRO

Recognized keys
Left, Right; Ctrl-Left, Ctrl-Right; Esc, F8, F10, Return, Home, End, Del, Ctrl-Backspace, Ctrl-Backspace, Ins

See also
lineinput

lineMenu

Predicate
lineMenu(posRow,Wattr,Fattr,ItemList,ChoiceCode)

Declaration
lineMenu(Row,Attr,Attr,StringList,Integer)

Flow pattern
lineMenu(i,i,i,i,o)

Function
The *lineMenu* predicate implements a pop-up horizontal line menu in which the arrow keys can be used to indicate a menu item; a selection is made by pressing F10 or Return.

Parameters
PosRow and *PosCol* define the upper left-hand corner of the window containing the menu. *Wattr* and *Fattr* determine the attributes for the window and its frame respectively. *ItemList* is a list of strings, one for each menu item. *Title* determines the text in the frame of the window. *ChoiceCode* becomes instantiated to an integer that specifies the selection made according to the following code:

0 = Esc was pressed during menu selection
1 = First menu item was selected
2 = Second menu item was selected

and so on.

Example
lineMenu(0,7,7,[Basic,Fortran,Apl],ch1)

Contained in
LINEMENU.PRO

Environment
TDOMS.PRO, TPREDS.PRO

lineShade

Predicate
lineShade(Row1,Col1,Row2,Col2,EndLine,FillColor,Direction)

Declaration
lineShade(VRow,VCol,VRow,VCol,VCol,Color,Kind) – (i,i,i,i,i,i,i) language c

Flow pattern
lineShade(i,i,i,i,i,i)

Function
Draws a line between two points given in virtual coordinates and shades up, down, left, or right from that line to a specified row or column of the graphics window.

Parameters
The line drawn is between the points (*Row1,Col1*) and (*Row2,Col2*), and an area on one side of the line is then shaded. The shading is up, down, left, or right depending on the *Direction* parameter:

0	Shade Up
1	Shade Down
2	Shade Left
3	Shade Right

The shading covers the area from the line drawn to the horizontal or vertical line indicated by the *EndLine* parameter. When shading is up or down, *Endline* represents a horizontal line (a Row coordinate must be given); when shading is to the left or right, *Endline* is taken to be the column coordinate for a vertical line.

Remarks
The predicate *LineShade* is implemented in assembler. When shading up or down, *Endline* is represented by a *VRow* value, and coercion between the domains *VRow* and *VCol* is necessary (take a look at the declaration of *lineShade*). This is done by a code fragment of the form

```
CoerceVariable = VRowvalue,lineShade(. . . . . . . .,CoerceVariable, . .),
```

Row1, Row2, Column1, Column2, and *EndLine* all must be in the range 0 to 31999.

Example
lineshade(1000,1000,1000,2000,2000,3,0)

Contained in
GRAPHICS.OBJ

Environment
GDOMS.PRO

loadpic

Predicate
*loadpic(FileName,StartRowPicFile,StartColPicFile,StartRowScreen,
StartColScreen,NoOfRows,NoOfCols)*

Declaration
*loadpic(String,Integer,Integer,Integer,Integer,Integer,Integer) –
(i,i,i,i,i,i) language c*

Flow pattern
loadpic(i,i,i,i,i,i)

Function
Loads a full-screen graphics image from a .PIC file created using PCPAINT (v1.0) or a program producing graphics-image files compatible with those produced by PCPAINT.

Parameters
Loads an image from the file determined by *Filename*. For the purpose of addressing various portions of this image, it is regarded as occupying 200 rows numbered 0 thru 199, and 320 columns, numbered 0 thru 319. *StartRowPicFile* and *StartColPicFile* specify the top left-hand corner of the sub-image to be selected from the full-screen image. *NoOfRows* and *NoOfCols* complete the specification of this sub-image. *StartRowScreen* and *StartColScreen* specify where on the screen the top left-hand corner of the sub-image is to be displayed.

Remarks
Works only in graphics mode 1. Requires an accompanying project definition.

Example
loadpic("demo.pic",50,80,0,0,100,160)

Contained in
PICTOOLS.OBJ

Environment
GDOMS.PRO, GGLOBS.PRO

See also
savepic

longmenu

Predicate
longmenu(Row,Col,Maxrows,Wattr,Fattr,ItemList,Title,InitItem,Choice)

Declaration
longmenu(Row,Col,Row,Attr,Attr,StringList,String,Integer,Integer)

Flow pattern
longmenu(i,i,i,i,i,i,i,i,o)

Function
The *longmenu* predicate implements a pop-up menu where the arrow keys and PgUp, PgDn, Home, and End can be used to indicate a menu item; the selection is actually made by pressing *F10* or Return. *longmenu* allows selection from over 23 choices.

Parameters
Row and *Col* determine the position of the window. *Wattr* and *Fattr* determine the attributes for the window and its frame respectively (if *Fattr* is zero, the window will not be framed). *Maxrows* determines how many rows should be displayed on the screen at any one time. *ItemList* lists the items on the menu. *Title* is the text at the top of the menu window. *InitItem* determines where the cursor is placed when the menu is first displayed. *Choice* becomes bound to the integer code for the selection made as follows:

0 = Esc was pressed
1 = First item was selected
2 = Second item was selected

and so on.

Remarks
The window is automatically adjusted if it is placed too far to the left or too near to the bottom of the screen. The predicate fails if the list of menu items has length zero.

Example
longmenu(5,5,7,7,5,[a,b,c,d,e,f,g,h,i,j,k,l,m,n],letters,0,CHOICE)

Contained in
LONGMENU.PRO

Environment
TDOMS.PRO, TPREDS.PRO

Recognized keys
Arrow keys, Esc, F10, Return, Home, End, PgUp, PgDn

See also
longmenu_leave, longmenu_mult, menu

longmenu_leave

Predicate
longmenu_leave(Row,Col,Maxrows,Wattr,Fattr,ItemList,Title,InitItem,Choice)

Declaration
longmenu_leave(Row,Col,Row,Attr,Attr,StringList,String,Integer,IntegerList)

Flow pattern
longmenu_leave(i,i,i,i,i,i,i,i,o)

Function
The *longmenu_leave* predicate implements a pop-up menu where the arrow keys, and PgUp, PgDn, Home, and End can be used to indicate a menu item; the selection is actually made by pressing F10 or Return. *longmenu_leave* allows selection from over 23 choices. After selection has been made, the menu window is left on the screen.

Parameters
Row and *Col* determine the position of the window. *Wattr* and *Fattr* determine the attributes for the window and its frame respectively (if *Fattr* is zero, the window will not be framed). *Maxrows* determines how many rows should be displayed on the screen at any one time. *ItemList* lists the items on the menu. *Title* is the text at the top of the menu window. *InitItem* determines where the cursor is placed when the menu is first displayed. *Choice* becomes bound to the integer code for the selection made as follows:

0 = Esc was pressed
1 = First item was selected
2 = Second item was selected

and so on.

Remarks
The window is automatically adjusted if it is placed too far to the left or too near the bottom of the screen. The predicate fails if the list of menu items has length zero.

Example
longmenu_leave(5,5,7,7,5,[a,b,c,d,e,f,g,h,i,j,k,l,m,n],letters,0,CHOICE)

Contained in
LONGMENU.PRO

Environment
TDOMS.PRO, TPREDS.PRO

Recognized keys
Arrow keys, Esc, F10, Return, Home, End, PgUp, PgDn

See also
longmenu, longmenu_mult, menu

longmenu_mult

Predicate
longmenu_mult(Row,Col,Maxrows,Wattr,Fattr,ItemList,Title,InitItems,ChoiceList)

Declaration
longmenu_mult(Row,Col,Row,Attr,Attr,StringList,String,Integer, List,
IntegerList)

Flow pattern
longmenu_mult(i,i,i,i,i,i,i,o)

Function
The *longmenu_mult* predicate implements a pop-up menu where the arrow keys and PgUp, PgDn, Home, and End can be used to indicate menu items; several selections are allowed, each selection being made by pressing F10 or Return. *longmenu_mult* allows multiple selections to be made from over 23 choices.

Parameters
Row and *Col* determine the position of the window. *Wattr* and *Fattr* determine the attributes for the window and its frame respectively (if *Fattr* is zero the window will not be framed). *Maxrows* determines how many rows should be displayed on the screen at any one time. *ItemList* lists the items on the menu. *Title* is the text at the top of the menu window.*InitItems* determines which entries should be highlighted when the menu is first displayed. and *ChoiceList* becomes bound to a list of the integer codes for the selections made as follows:

0 = Esc was pressed
1 = First item was selected
2 = Second item was selected

and so on.

Remarks
The window is automatically adjusted if it is placed too far to the left or too near to the bottom of the screen. The predicate fails if the list of menu items has length zero.

Example
longmenu_mult(5,5,7,7,5,[a,b,c,d,e,f,g,h,i,j,k,l,m,n],letters,0,ChoiceList)

Contained in
LONGMENU.PRO

Environment
TDOMS.PRO, TPREDS.PRO

Recognized keys
Arrow keys, Esc, F10, Return, Home, End, PgUp, PgDn

See also
longmenu, longmenu_leave, menu

makeAxes

Predicate
*makeAxes(AxesNo,AxesWindowNo,GraphWindow,Xmarkers,Ymarkers,
Left,Bottom,Right,Top)*

Declaration
makeAxes(Integer,Integer,Integer,XMarker,YMarker,Col,Row,Col,Row)

Flow pattern
makeAxes(i,o,o,i,i,i,i,i,i)

Function
makeAxes is used to draw a pair of axes.

Parameters
AxesNo is bound to a value that the programmer can use to refer to a particular pair of axes. *AxesWindowNo* records in which window the axes are drawn. When defining axes using *makeAxes,* the first step is to draw the axes in the currently active window. Then a window is created inside that active window. The number of this window is returned in *GraphWindowNo.* Then, subsequent images drawn in that coordinate system are automatically clipped when the image lies outside the range for the active scale.

The markings on the axes are defined by an *XMarker* and a *YMarker,* both of which take the form

```
marker(Unit,FormatSpecifier,FieldWidth)
```

This defines the interval *Unit* at which markings are to appear on the relevant axis; whether the number marked is to appear in decimal form or exponential form (*FormatSpecifier* equal to *d* or *e* respectively); and, finally, the *FieldWidth* in which the numeric markings are to be displayed.

The last four parameters in *makeAxes* determine how much space should be left between the borders of the window and the edge of the graphics screen. All of these are given as text coordinates.

Example
makeAxes(1,_,_,marker(10,d,2),marker(10,d,3),2,3,2,1)

Contained in
GGRAPH.PRO

Environment
TDOMS.PRO, GDOMS.PRO, TPREDS.PRO, GPREDS.PRO, GGRAPH.PRO

See also
refreshAxes, modifyAxes, labelAxes

makestatus

Predicate
makestatus(StatusWindowAttr, StringEntry)

Declaration
makestatus(Attr, String)

Flow pattern
makestatus(i, i)

Function
Displays a status line at the bottom of the screen.

Parameters
The text to which *StringEntry* is bound is displayed in the status line that has an attribute given by *StatusWindowAttr*.

Remarks
Uses window number 83 for the status line.

Example
makestatus("d = date   t = time   m = more   l = less")

Contained in
STATUS.PRO

Environment
TDOMS.PRO, TPREDS.PRO

See also
refreshstatus, removestatus, tempstatus

menu

Predicate
menu(PosRow,PosCol,Wattr,Fattr,ItemList,Title,InitItem,ChoiceCode)

Declaration
menu(Row,Col,Attr,Attr,StringList,String,Integer,Integer)

Flow pattern
menu(i,i,i,i,i,i,o)

Function
The *menu* predicate implements a pop-up menu in which the arrow keys can be used to indicate a menu item; a selection is made by pressing F10 or Return.

Parameters
PosRow and *PosCol* define the upper left-hand corner of the window containing the menu. *Wattr* and *Fattr* determine the attributes for the window and its frame respectively. *ItemList* is a list of strings, one for each menu item. *Title* determines the text in the frame of the window. *InitItem* specifies which menu item the cursor is to highlight when the menu is first displayed, and *ChoiceCode* becomes instantiated to an integer that specifies the selection made according to the following code:

0 = Esc was pressed during menu selection
1 = First menu item was selected
2 = Second menu item was selected

and so on.

Remarks
The window is automatically adjusted if it is placed too far to the left or too near to the bottom of the screen. The predicate fails if the list of menu items has length zero. If the number of menu items is greater than the number of available rows within the containing window, a run-time error results.

Example
menu(5,10,7,7,[basic,pascal,lisp,prolog],"Language",2,ChoiceCode)

Contained in
MENU.PRO

Environment
TDOMS.PRO, TPREDS.PRO

Recognized keys
Arrow keys, Esc, F10, Return, Home, End, PgUp, PgDn

See also
longmenu, menu_leave, menu_mult

menu_leave

Predicate
menu_leave(PosRow,PosCol,Wattr,Fattr,ItemList,Title,InitItem,ChoiceCode)

Declaration
menu_leave(Row,Col,Attr,Attr,StringList,String,Integer,Integer)

Flow pattern
menu_leave(i,i,i,i,i,i,i,o)

Function
The *menu_leave* predicate implements a pop-up menu in which the arrow keys can be used to indicate a menu item; a selection is made by pressing F10 or Return. Unlike *menu,* however, *menu_leave* does not remove the menu from the screen after a choice has been made.

Parameters
PosRow and *PosCol* define the upper left-hand corner of the window containing the menu. *Wattr* and *Fattr* determine the attributes for the window and its frame respectively. *ItemList* is a list of strings, one for each menu item. *Title* determines the text in the frame of the window. *InitItem* specifies which menu item the cursor is to highlight when the menu is first displayed, and *ChoiceCode* becomes instantiated to an integer that specifies the selection made according to the following code:

0 = Esc was pressed during menu selection
1 = First menu item was selected
2 = Second menu item was selected

and so on.

Remarks
The window is automatically adjusted if it is placed too far to the left or too near to the bottom of the screen. The predicate fails if the list of menu items has length zero. If the number of menu items is greater than the number of available rows within the containing window, a run-time error results.

Example
menu_leave(5,10,7,7,[basic,pascal,lisp,prolog],"Language",2,ChoiceCode)

Contained in
MENU.PRO

Environment
TDOMS.PRO, TPREDS.PRO

Recognized keys
Arrow keys, Esc, F10, Return, Home, End, PgUp, PgDn

See also
longmenu, menu, menu_mult

menu_mult

Predicate
menu_mult(PosRow,PosCol,Wattr,Fattr,ItemList,Title,InitItems,ChoiceList)

Declaration
menu_mult(Row,Col,Attr,Attr,StringList,String,IntegerList,IntegerList)

Flow pattern
menu_mult(i,i,i,i,i,i,i,o)

Function
The *menu_mult* predicate implements a pop-up menu from which multiple choices can be made. The arrow keys are used to indicate a menu item, and each selection is made by pressing Return. The user presses F10 once all selections have been made.

Parameters
PosRow and *PosCol* define the upper left hand corner of the window containing the menu. *Wattr* and *Fattr* determine the attributes for the window and its frame respectively. *ItemList* is a list of strings, one for each menu item. *Title* determines the text in the frame of the window. *InitItem* specifies which menu items should be highlighted when the menu is first displayed, and *ChoiceList* becomes instantiated to a list of integers that specify the selection made according to the following code:

0 = Esc was pressed during menu selection
1 = First menu item was selected
2 = Second menu item was selected

and so on.

Remarks
The window is automatically adjusted if it is placed too far to the left or too near to the bottom of the screen. The predicate fails if the list of menu items has length zero. If the number of menu items is greater than the number of available rows within the containing window, a run-time error results.

Example
menu_mult(5,10,7,7,[basic,pascal,lisp,prolog],"Language",[2],ChoiceList)

Contained in
MENU.PRO

Environment
TDOMS.PRO, TPREDS.PRO

Recognized keys
Arrow keys, Esc, F10, Return, Home, End, PgUp, PgDn

See also
longmenu, menu, menu_leave

mkdir

Predicate
mkdir(NewSubDir)

Declaration
mkdir(String)

Flow pattern
mkdir(i)

Function
mkdir creates a new subdirectory corresponding to the DOS function MKDIR.

Parameters
NewSubDir should be bound to a string containing a maximum of eight letters; a new directory is created with that name.

Remarks
This predicate is also described in the *Turbo Prolog Owner's Handbook.*

Example
mkdir(samples)

Contained in
BIOS.PRO

See also
rmdir

modifyAxes

Predicate
modifyAxes(AxesNo,MyXmarker,MyYmarker)

Declaration
modifyAxes(Integer,XMarker,YMarker)

Flow pattern
modifyAxes(i,i,i)

Function
Used to modify (the markings on) a set of axes.

Parameters
AxesNo identifies the pair of axes to be affected and is the axes identifier returned from a previous call of *makeAxes*. *MyXmarker* and *MyYmarker* both have the following format:

```
marker(Unit,FormatSpecifier,FieldWidth)
```

This defines the interval *Unit* at which markings are to appear on the relevant axis; whether the number marked is to appear in decimal form or exponential form (*FormatSpecifier* equal to *d* or *e* repectively); and, finally, the *FieldWidth* in which the numeric markings are to be displayed.

Example
modifyAxes(2,marker(10,d,4),marker(10,d,5))

Contained in
GGRAPH.PRO

Environment
TDOMS.PRO, GDOMS.PRO, TPREDS.PRO, GPREDS.PRO

See also
makeAxes, refreshAxes

noinput (programmer defined)

Predicate
noinput(MyFieldName)

Declaration
noinput(FName)

Flow pattern
noinput(i)

Function
Called by *scrhnd* in conjunction with a screen definition created using SCRDEF.PRO, to determine those display fields in which the user can't enter information.

Parameters
Declares the field to which *MyFieldName* is bound as one in which the user is not allowed to type.

Remarks
A default defining clause for *noinput* is given in HNDBASIS.PRO (see "No Input Fields" in Chapter 3). Other defining clauses must be added by the programmer.

Example
noinput(clock)

Contained in
SCRHND, VSCRHND

Environment
TPREDS.PRO, TDOMS.PRO, LINEINP.PRO, STATUS.PRO

See also
scrhnd

openRS232

Predicate
openRS232(PortNo, InputBufSize, OutputBufSize, BaudRate,
Parity, WordLength, StopBits, Protocol)

Declaration
openRS232(Integer,Integer,Integer,Integer,Integer,Integer,Integer,Integer)
– (i,i,i,i,i,i,i,i) language c

Flow pattern
openRS232(i,i,i,i,i,i,i,i)

Function
The tool predicate *openRS232* initializes either COM1 or COM2 ready for transmitting or receiving data. It fails if either the Asynchronous Adapter (or equivalent) is not in the PC or one of its parameters is out of range (for example, if an illegal value is given for baud rate or I/O port number).

Parameters

PortNo	= 1 means use COM1. = 2 means use COM2.
InputBufSize	must be in the range 1 to 31767 and specifies the number of bytes reserved for the input buffer.
OutputBufSize	must be in the range 1 to 31767 and specifies the number of bytes reserved for the output buffer.
BaudRate	= 0 means 110 Baud. = 1 means 150 Baud. = 2 means 300 Baud. = 3 means 600 Baud. = 4 means 1200 Baud. = 5 means 2400 Baud. = 6 means 4800 Baud. = 7 means 9600 Baud.
Parity	= 0 means no parity. = 1 means odd parity. = 2 means even parity.

WordLength	= 0 means 5 data bits. = 1 means 6 data bits. = 2 means 7 data bits. = 3 means 8 data bits.
StopBits	= 0 means 1 stop bit. = 1 means 2 stop bits.
Protocol	= 0 means communication with *neither* XON/XOFF *nor* RTS/CTS. = 1 means communication *with* XON/XOFF but *without* RTS/CTS (the preferred mode). = 2 means communication *with* RTS/CTS but *without* XON/XOFF. If RTS (Request To Send) is high, then CTS will go high when the external device is ready to receive (and vice-versa). = 3 means communication with *either* XON/XOFF *or* RTS/CTS.

Remarks
Correct hardware connection must have been established (see Chapter 5). Requires the creation of a project.

Example
openRS232(1, 256, 256, 7, 0, 3, 0, 2)

Contained in
SERIAL.OBJ

Environment
CONGLOBS.PRO, TICKS.OBJ

See also
closeRS232

pdwaction (programmer defined)

Predicate
pdwaction(HorizontalCode, VerticalCode)

Declaration
pdwaction(Integer,Integer)

Flow pattern
pdwaction(i,i)

Function
In a pull-down menu system built using *pulldown,* the action taken when a menu item has been finally selected from all the various categories and subcategories is determined by the *pdwaction* predicate. This allows the program to perform actions while the pull-down window remains on the screen. If *pdwaction* fails, then *Choice* and *SubChoice* in *pulldown* (see the separate entry) are bound to the relevant values, the entire menu system is removed from the screen, and *pulldown* succeeds. If, on the other hand, *pdwaction* succeeds, then *pulldown* continues to loop and the last selected menu item remains highlighted.

Parameters
HorizontalCode and *VerticalCode* determine the menu item for which the actions described on the right-hand side of a given clause definition for *pdwaction* will be invoked.

Remarks
To use *pulldown* you must include the **database** predicate *pdwstate* in your program's **database** declarations as follows:

```
database
   pdwstate(Row,Col,SYMBOL,Row,Col)
```

Example
pdwaction(5,2) :- dir("",".*",X),....*

Contained in
PULLDOWN.PRO

Environment
TDOMS.PRO, TPREDS.PRO

See also
pulldown

pieChart

Predicate
pieChart(Row,Col,Radius,PieSegmentList)

Declaration
pieChart(VRow,VCol,VRadius,PieSegmentList)

Flow pattern
pieChart(i,i,i,i)

Function
Used to display a list of values (percentages) as pie slices, so they make up a whole pie; that is, the slices are adjusted so the total is 100 percent.

Parameters
(*Row,Col*) are the (virtual) coordinates of the center of the pie, which has the given *Radius* (also expressed as a virtual coordinate value). *PieSegmentList* is a list of pie slices each specified by:

- A percentage value: A negative percentage causes the slice to be moved outwards, thus drawing attention to that particular slice.
- An optional label: If the label ends with an = character, the label is suffixed with the actual percentage value.
- A fill color value: A zero color value indicates that only the frame of the slice is to be colored; a positive value causes the slice to be filled.
- A frame color value: This color is used for the label, too.

Remarks
The domain *PIESEGMENTLIST* is defined in GPIE.PRO as follows:

```
PERCENT             = REAL
PIESEGMENT          = slice(PERCENT,String,Color,Color);space
PIESEGMENTLIST      = PIESEGMENT*
```

It requires the creation of a project.

Example

```
pieChart(15000,20000,3000,
          [slice(16,"May",1,2),
           slice(10.7,"Jun",2,3),
           slice(15.1,"Jul",3,2),
           slice(6,"Aug",2,1)]),
```

Contained in
GPIE.PRO

Environment
TDOMS.PRO. TPREDS, GDOMS.PRO, GPREDS.PRO, GGLOBS.PRO, GRAPHICS.OBJ

See also
Piechart

plot

Predicate
plot(Row,Col,Color,Size,PlotKind)

Declaration
plot(VRow,VCol,Color,Size,Kind) – (i,i,i,i,i) language c

Flow pattern
plot(i,i,i,i,i)

Function
Plots a point specified by virtual coordinates in the graphics window, using different shapes.

Parameters
The virtual coordinates (*Row, Col*) specify the point with the given *Color*. The size of the marking is determined by the parameters *Size,* using virtual coordinates. *PlotKind* determines the shape plotted as follows:

0 = One pixel
1 = A dot is drawn inside a box
2 = A box is drawn and filled
3 = An X is drawn

Remarks
The predicate is implemented in C.

Range for the *Row* parameter: 0 to 31999
Range for the *Col* parameter: 0 to 31999
Range for the *Size* parameter: 0 to 31999

It requires the creation of a project.

Example
plot(16000,16000,15,1000,3)

Contained in
GRAPHICS.OBJ

Environment
GDOMS.PRO, GGLOBS.PRO

pop_helpcontext

Predicate
pop_helpcontext

Declaration
pop_helpcontext

Flow pattern
None, takes no parameters.

Function
When a program leaves the current context, the tool predicate *pop_helpcontext* is used to remove the current *helpcontext* and re-establish the old.

Parameters
None

Example
pop_helpcontext

Contained in
HELP.PRO

Environment
TDOMS.PRO, TPREDS.PRO

See also
help, helptext, push_helpcontext

pulldown

Predicate
pulldown(AttForAllWindows,MenuList,Choice,Subchoice)

Declaration
pulldown(Attr,MenuList,Integer,Integer)

where MenuList is defined by

```
MENUELEM = curtain(Col,String,StringList)
MENULIST = MENUELEM*
```

Flow pattern
pulldown(i,i,o,o)

Function
pulldown allows a menu in which items can be grouped into related families, as in the Turbo Prolog user interface. Once a choice has been made from a horizontally displayed menu (Run, Compile, Edit, and so on), a second menu is pulled down vertically below that horizontal menu item. This second menu contains items closely related to the horizontal heading.

Parameters
AttForAllWindows is the attribute to be used in all the windows (and their frames) forming the pull-down menu system. *Menulist* is a list constructed using the *curtain* functor, which contains the text for each of the menus that can be pulled down. On return from *pulldown, Choice* is bound to the code for the horizontal-menu selection; on return from *pulldown, Subchoice* is bound to the code for the vertical-menu selection, except that if there is no vertical menu corresponding to a horizontal menu item, *Subchoice* is bound to 0. *Menulist* is specified by giving a list of values for the *curtain* functor. It takes three parameters:

curtain(Col,String,StringList)

The first specifies which column the horizontal menu item should commence on, the second gives the name of that menu item, and the third lists the items that are to appear in the corresponding vertical menu. Thus,

curtain(4,"Animals",["Dog","Cat","Bullfinch"])

specifies part of a pull-down menu system in which, if the *Animals* heading is selected, a vertical menu appears containing *Dog,Cat,* and *Bullfinch.*

Remarks
When a choice has been made from a *pulldown* menu, any actions to be carried out must be specified by clauses for the *pdwaction* predicate.

Example
```
pulldown(7,[curtain(3,"Input",["First","Second","Third"]),
            curtain(14,"List",[]),
            curtain(23,"Files",["Load","Save","Delete","Directory"]),
            curtain(35,"Setup",["Directories","Colors"]),
            curtain(46,"Quit" ,[])]
         ,CH,SUBCH )
```

Contained in
PULLDOWN.PRO

Environment
TDOMS.PRO, TPREDS.PRO

See also
pdwaction

push_helpcontext

Predicate
push_helpcontext(NameOfHelpContext)

Declaration
push_helpcontext(HelpContext)

Flow pattern
push_helpcontext(i)

Function
When a program moves into a new context, the tool predicate *push_helpcontext* can be used to push the name of a new *helpcontext* onto the stack.

Parameters
Enables the Help predicate to access the Help text known as *NameOfHelpContext*, created by HELPDEF.PRO.

Remarks
HELPDEF.PRO is described in "Context-Sensitive Help," Chapter 2.

Example
push_helpcontext(submenu1)

Contained in
HELP.PRO

Environment
TDOMS.PRO, TPREDS.PRO

See also
help, pop_helpcontext

queueSize

Predicate
queueSize(PortNo, SizeOfInputQueue, SizeOfOutputQueue)

Declaration
queueSize_RS232(Integer,Integer,Integer) – (i,o,o) language c

Flow pattern
queueSize_RS232(i,o,o)

Function
queueSize_RS232 returns the size of the input and output queues: The first contains the characters that the low-level routines have received and that haven't been read yet by the calling Turbo Prolog program; the second holds the characters the Turbo Prolog program has written but that haven't been transmitted yet. *queueSize* fails if the specified COM port is not open.

Parameters
With *PortNo* bound to the code for a serial I/O port (*PortNo=1* means COM1, *PortNo*=2 means COM2), *SizeOfInputQueue* and *SizeOfOutputQueue* become bound to the number of characters in the input and output queues, respectively.

Remarks
Requires the definition of a project.

Example
queue_RS232(2, X, Y)

Contained in
SERIAL.OBJ

Environment
COMGLOBS.PRO, TICKS.OBJ

rd_dBase3File

Predicate
rd_dBase3File(TotRecs,File,FldDescL,dBase3RecL)

Declaration
rd_dBase3File(Real,File,FldDescL,dBase3RecL)

Flow pattern
rd_dBase3File(i,i,i,o)

Function
Using the tool predicate *rd_dBase3File,* the dBASE III data records can be read and collected in a list belonging to the tool domain DBASE3RECL, which has the following declaration:

```
DBASE3RECL = DBASE3REC*                          /* The database is a list of records*/
DBASE3REC  = DBASE3ELEM*                               /* Fields in each records */
DBASE3ELEM = char(String);                                        /* Characters */
             real(REAL);                           /* 64-bit IEEE floating point */
             logical(BOOL);                                          /* Logical */
             memo(String);                     /* Memo text loaded from .DBT file */
             date(String)                                  /* Format YYYY MM DD */
BOOL       = CHAR                                      /* Y y N n T t F f or Space */
```

Remarks
Requires the creation of a project.

Contained in
DBASE3.PRO

Environment
TDOMS.PRO, TPREDS.PRO, DBASE3.PRO, READEXT.PRO

See also
init_dBase3, rd_dBase3Rec

rd_dBase3Rec

Predicate
rd_dBase3Rec(TotRecs,File,FldDescL,dBase3Rec)

Declaration
rd_dBase3Rec(Real,File,FldDescL,dBase3Rec)

Flow pattern
rd_dBase3Rec(i,i,i,o)

Function
The tool predicate *rd_dBase3Rec* allows you to read one record at a time of a dBASE III file, the same way you access a Reflex file record by record. Thus, using *rd_dBase3Rec,* it is possible to read a record, do some computation, remove the storage allocation via backtracking, and then read the next record.

```
DBASE3REC  = DBASE3ELEM*                              /* Fields in each record */
DBASE3ELEM = char(String);                                   /* Characters */
             real(REAL);                        /* 64-bit IEEE floating point */
             logical(BOOL);                                     /* Logical */
             memo(String);                 /* Memo text loaded from .DBT file */
             date(String)                            /* Format YYYY MM DD */

BOOL       = CHAR                                  /* Y y N n T t F f or Space */
```

Remarks
Requires the creation of a project.

Contained in
DBASE3.PRO

Environment
TDOMS.PRO, TPREDS.PRO, DBASE3.PRO, READEXT.PRO, REALINTS.OBJ

rd_LotusCell

Predicate
rd_LotusCell(LotusRec)

Declaration
rd_LotusCell(LotusRec)

Flow pattern
rd_LotusCell(Rec) – (i),(o)
rd_LotusCell(Elem(Row,Col,Value)) – (i,i,i),(i,i,o),...,(o,o,o).

Function
The tool predicate *rd_LotusCell* is used to search for an instantiated cell in the spreadsheet file.

Remarks
Requires the creation of a project.

Contained in
LOTUS.PRO

Environment
TDOMS.PRO, TPREDS.PRO, LOTUS.PRO, READEXT.PRO, REALINTS.OBJ

See also
rd_LotusFile

rd_LotusFile

Predicate
rd_LotusFile(LotusRecL)

Declaration
rd_LotusFile(LotusRecL)

Flow pattern
rd_LotusFile(o)

Function
The tool predicate *rd_LotusFile* is used for reading all cells in a spreadsheet into a data structure.

Parameters

```
LOTUSRECL = LOTUSREC*

LOTUSREC  = version(Integer);                               /* Version number */
            elem(Integer,Integer,VALUE);

VALUE     = int(Integer);                             /* Integer number cell */
            real(REAL);                                  /* Real number cell */
            formula(REAL);                           /* Defines a formula cell */
            label(String);                            /* Defines a label cell */]
```

Remarks
Requires the creation of a project.

Contained in
LOTUS.PRO

Environment
TDOMS.PRO, TPREDS.PRO, LOTUS.PRO, READEXT.PRO, REALINTS.OBJ

See also
rd_LotusCell

rd_ReflexFile

Predicate
rd_ReflexFile(TotRecs,ReflexTypeL,TextPools,ReflexRecL)

Declaration
rd_ReflexFile(Integer,ReflexTypeL,TextPools,ReflexRecL)

Flow pattern
rd_ReflexFile(i,i,i,o)

Function
Using the tool predicate *rd_ReflexFile,* all the Reflex data records in a given file can be read and collected into a single Turbo Prolog list at the same time. This list belongs to the tool domain REFLEXRECL which is declared as follows:

```
REFLEXRECL = REFLEXREC*                    /* The database is a list of records */
REFLEXREC  = REFLEXELEM*                   /* A record is a list of elements */
REFLEXELEM = date(Integer);                /* 16-bit int. representing number of*/
                                           /* days since December 31, 1899 */
             real(REAL);                   /* 64-bit IEEE floating point real */
             int(Integer);                 /* 16-bit signed integer */
             text(String);                 /* A string representing a text */
             untyped;                      /* No data stored */
             error
```

Parameters
The parameters *TotRecs, ReflexTypeL,* and *TextPools* are the values returned by the *init_Reflex* predicate.

Contained in
REFLEX.PRO

Environment
TDOMS.PRO, TPREDS.PRO, REFLEX.PRO, READEXT.PRO, REALINTS.OBJ

See also
Init_Reflex, rd_ReflexRec

rd_ReflexRec

Predicate
rd_ReflexRec(TotRecs,ReflexTypL,TextPools,ReflexRec)

Declaration
rd_ReflexRec(Integer,ReflexTypL,TextPools,ReflexRec)

Flow pattern
rd_ReflexRec(i,i,i,o)

Function
The tool predicate *rd_ReflexRec* allows access to one Reflex record at a time. In order to access Reflex records, it is first necessary to call the tool predicate *init_Reflex*.

Parameters
The parameter REFLEXREC belongs to the domain:

```
REFLEXREC  = REFLEXELEM*                              /* A record is a list of elements */
REFLEXELEM = date(Integer);                          /* 16-bit int. representing number of
                                                     /* days since December 31, 1899 */
             real(REAL);                              /* 64-bit IEEE floating-point real */
             int(Integer);                                        /* 16-bit signed integer */
             text(String);                                   /* A string representing a text */
             untyped;                                                     /* No data stored */
             error
```

Remarks
Requires the creation of a project.

Contained in
REFLEX.PRO

Environment
TDOMS.PRO, TPREDS.PRO, REFLEX.PRO, READEXT.PRO, REALINTS.OBJ

See also
init_Reflex, rd_ReflexFile

readfilename

Predicate
readfilename(Row,Col,Wattr,Fattr,Extension,OldFileName,NewFileName)

Declaration
readfilename(Row,Col,Attr,Attr,String,String,String)

Flow pattern
readfilename(i,i,i,i,i,i,o)

Function
Implements a file-name input facility similar to that in the Turbo prolog user interface. Inside a window, the user can type just the file's first name and have a default file type added—just as Turbo Prolog adds the .PRO extension to file names when LOADing and SAVEing programs. Alternatively, if the user presses Return with the window empty, then a directory appears in a window on the screen, from which the user can select a file name by using the arrow keys and pressing Return, as with Turbo Prolog.

Parameters
Row and *Col* determine the position of the input field on the screen. *Wattr* and *Fattr* are the window and frame attributes respectively. *Extension* is the String that is to be appended to the file name if no file type is specified. *OldFileName* is the file name to be displayed in the window when it is first displayed. The user can edit this text or leave it unchanged, then press Return or F10 once the required file name is in the window. *NewFileName* becomes bound to the new file name, with the *Extension* added automatically to the user's input if no file type is specified.

Remarks
A program that uses *readfilename* must contain the following **database** declarations at an appropriate point:

```
database
   insmode
   lineinpstate(STRING,COL)
   lineinpflag
```

Example
readfilename(10,10,7,7,pro,"oldname.dat",NewName)

Contained in
FILENAME.PRO

Environment
TDOMS.PRO, TPREDS.PRO, LINEINP.PRO

refreshstatus

Predicate
refreshstatus

Declaration
refreshstatus

Flow pattern
None, takes no parameters.

Function
Refreshes a status line created by *makestatus*. In particular, it can be used to pull the status window to the foreground if window resizing has moved it into the background.

Remarks
Uses window 83 for the status line.

Example
refreshstatus

Contained in
STATUS.PRO

Environment
TDOMS.PRO, TPREDS.PRO

See also
makestatus, removestatus, tempstatus

removestatus

Predicate
removestatus

Declaration
removestatus

Flow pattern
None, takes no parameters.

Function
Deletes a status line created by *makestatus*.

Remarks
Releases window number 83 from use as a status line.

Example
removestatus

Contained in
STATUS.PRO

Environment
TDOMS.PRO, TPREDS.PRO

See also
makestatus, refreshstatus, tempstatus

resizewindow

Predicate
resizewindow

Declaration
resizewindow

Flow pattern
None, takes no parameters.

Function
Allows the currently active window to be resized, using the arrow keys for small increments and decrements in size and Ctrl with the arrow keys for larger steps; Shift plus the arrow keys move the window. Home, End, PgUp, and PgDn can also be used to move and resize the window. After resizing the values of the maximum Row and Column occupied by the window, they are saved in the *windowsize* **database**.

Remarks
The declaration of the *windowsize* **database** is

```
database
  windowsize(ROW,COL)
```

Each time a window is changed using resize, *writescr* is called. You must define this predicate so that the screen is updated appropriately.

Example
resize

Contained in
RESIZE.PRO

Environment
STATUS.PRO, TDOMS.PRO, TPREDS.PRO

See also
status, writescr

rmdir

Predicate
rmdir(SubDir)

Declaration
rmdir(String)

Flow pattern None, takes no parameters.

Function
rmdir removes a subdirectory, functioning like the DOS RMDIR command.

Parameters
Subdir must be bound to the name of an existing subdirectory.

Remarks
rmdir is also described in the *Turbo Prolog Owner's Handbook.*

Example
rmdir(examples)

Contained in
BIOS.PRO

See also
mkdir

rxch_RS232

Predicate
rxch_RS232(PortNo,CH)

Declaration
rxch_RS232(Integer,Char)

Flow pattern
rxch_RS232(i,o)

Function
rxch_RS232 returns a character from the input buffer if there are any left.

Parameters
Binds *CH* to the next available character (if any) from the input buffer for port number *PortNo*, where *PortNo=1* means the COM1 communication port and *PortNo*=2 means the COM2 communication port.

Remarks
rxch_RS232 fails if the input buffer is empty or the specified port is not open. If anything goes wrong, more information can be obtained by calling the *status_RS232* tool predicate to obtain more information about the current transmission state.

Requires a project definition.

Example
rxch_RS232(1,InChar)

Contained in
SERIAL.OBJ

Environment
COMGLOBS.PRO, TICKS.OBJ

See also
status_RS232

rxStr_Modem

Predicate
rxStr_Modem(TextReceived)

Declaration
rxStr_Modem(String) – (o) language c

Flow pattern
rxStr_Modem(o)

Function
Takes a single string parameter that is bound to the characters received (if any) from the remote modem. If there is a Command Terminator (see *setModemMode*) in the input buffer or any preceding ATtention character, the returned string won't include these extra characters unless the Command Terminator is the NULL character ('\000'). In this case, everything in the input buffer is returned in the string parameter. *rxStr_Modem* fails if a serial I/O port has not been opened for the modem.

Parameters
TextReceived is bound to the characters received (if any) from the remote modem.

Remarks
Requires the definition of a project.

Example
rxStr_Modem(MyString)

Contained in
MODEM.OBJ

Environment
COMGLOBS.PRO

See also
SetModemMode

savepic

Predicate
savepic(DosFileName)

Declaration
savepic(String) – (i) language c

Flow pattern
savepic(i)

Function
A graphics screen image is saved in a disk file and works in graphics mode 1 only.

Parameters
DosFileName determines the file name under which the image is stored.

Remarks
Requires an accompanying project definition

Example
savepic("PRETTY.PIC")

Contained in
PICTOOLS.OBJ

Environment
TDOMS.PRO, TPREDS.PRO, GDOMS.PRO, GGLOBS.PRO

See also
loadpic

scaleCursor

Predicate
scaleCursor(XscaleCoord,YscaleCoord)

Declaration
scaleCursor(X,Y)

Flow pattern
scaleCursor(i,i)

Function
The predicate *scaleCursor* is used to position the cursor when working with scaled graphics.

Parameters
The cursor is positioned at scale coordinate *[XscaleCoord,YscaleCoord]*.

Example
scaleCursor(100,100)

Contained in
GGRAPH.PRO

Environment
TDOMS.PRO, GDOMS.PRO, TPREDS.PRO, GPREDS.PRO

See also
draw, scaleLine, scalePlot, scalePolygon

scaleLine

Predicate
scaleLine(X1,Y1,X2,Y2,Color)

Declaration
scaleLine(X,Y,X,Y,Color)

Flow pattern
scaleLine(i,i,i,i,i)

Function
Used to plot lines when working with a scaled coordinate system.

Parameters
(*X1,Y1*) and (*X2,Y2*) are the points represented in scaled coordinates that the line is to join. The line is drawn in the given *Color*.

Example
scaleLine(5,5,50,50,3)

Contained in
GGRAPH.PRO

Environment
TDOMS.PRO, GDOMS.PRO, TPREDS.PRO, GPREDS.PRO

See also
scalePlot, scaleCursor, scalePolygon

scalePlot

Predicate
scalePlot(X,Y,Color)

Declaration
scalePlot(X,Y,Color)

Flow pattern
scalePlot(i,i,i)

Function
The predicate *scalePlot* is used to plot points when a scaled coordinate system is in use.

Parameters
[X,Y] defines a point in the scaled coordinate system that is plotted in the given *Color*.

Remarks
The drawing speed can be increased by declaring the domains *X* and *Y* as INTEGERs instead of REALs.

Example
scalePlot(100,200,3)

Contained in
GGRAPH.PRO

Environment
TDOMS.PRO, GDOMS.PRO, TPREDS.PRO, GPREDS.PRO

See also
scaleLine, scaleCursor, scalePolygon, draw

scalePolygon

Predicate
scalePolygon(MyColor,MyDrawing)

Declaration
scalePolygon(Color,Drawing)

Flow pattern
scalePolygon(i,i)

Function
Draws a polygon using scaled coordinates.

Parameters
A polygon is drawn in the given *Color*. The parameter *MyDrawing* should belong to the domain DRAWING, which is declared as

```
X,Y = REAL
POINT = p(X,Y)
DRAWING = POINT*
```

Remarks
The drawing speed can increased by declaring the domains *X* and *Y* as INTEGERS instead of REALs.

Example
scalePolygon(3,[p(5,5),p(5,10),p(10,10),p(5,5)])

Contained in
GGRAPH.PRO

Environment
TDOMS.PRO, GDOMS.PRO, TPREDS.PRO, GPREDS.PRO

scale_text

Predicate
scale_text(MyScaleX,MyScaleY,MyTextRow,MyTextCol)

Declaration
scale_text(X,Y,Row,Col)

Flow pattern
scale_text(i,i,o,o)

Function
Converts between scaled coordinates and text coordinates in a graphics window.

Parameters
[MyScaleX,MyScaleY] are the scaled coordinates; *[MyTextRow,MyTextCol]* are the text coordinates.

Example
scale_text(MyX,MyY,0,0)

Contained in
GGRAPH.PRO

Environment
TDOMS.PRO, GDOMS.PRO, TPREDS.PRO, GPREDS.PRO

scale_virtual

Predicate
scale_virtual(MyScaleX,MyScaleY,MyVirtualRow,MyVirtualCol)

Declaration
scale_virtual(X,Y,VRow,VCol)

Flow pattern
scale_virtual(i,i,o,o)

Function
Converts between scaled and virtual coordinates.

Parameters
[MyScaleX,MyScaleY] are the scaled coordinates;
[MyVirtualRow,MyVirtualCol] are the virtual coordinates.

Example
scale_virtual(10,20,CorrespVrow,CorrespVcol)

Contained in
GGRAPH.PRO

Environment
TDOMS.PRO, GDOMS.PRO, TPREDS.PRO, GPREDS.PRO, GGRAPH.PRO

scrhnd

Predicate
scrhnd(TopLineSwitch, KeyUsedForReturn)

Declaration
scrhnd(Symbol,Key)

Flow pattern
scrhnd(i,o)

Function
Creates a screen display from a screen-layout definition constructed using SCRDEF.PRO (which is described in "Facilities in SCRDEF.PRO" in Chapter 3). It also enables fields in a layout to be defined as *action, value,* or *noinput* fields via the predicates *field_action, field_value,* and *noinput* respectively.

Parameters
scrhnd is used as part of a sequence of calls similar to the following:

```
   consult("MYLAYOUT.SCR"),
   createwindow(TopLineSwitch),
   scrhnd(TopLineSwitch, KeyUsedForReturn)
```

They bring *scrhnd* into action on the screen layout definition in MYLAYOUT.SCR.

scrhnd uses the values **consulted** from MYLAYOUT.SCR to form a screen display conforming to the layout specified. The application's user can then fill in the entries for each field and re-edit them in any order until he or she terminates input by pressing F10 or Esc. The terminating key is returned in the parameter *KeyUsedForReturn*. During data input with the screen layout in MYLAYOUT.SCR, if *TopLineSwitch* is bound to ON, a line is displayed at the top of the screen giving the name of the field that currently contains the cursor. No such line is displayed if it is bound to OFF.

Remarks
Any containing program must declare the following domains and **database** predicates. For ease of use, these and the basic framework for the definition of *field_action, field_value,* and *noinput* have been collected together in *HNDBASIS.PRO.*

```
domains
```

```
domains
   FNAME = SYMBOL
   TYPE = int(); str(); real()

database
/* database declarations used in SCRHND */
   insmode                                              /* Global insertmode */
   actfield(FNAME)                                           /* Actual field */
   screen(SYMBOL,DBASEDOM)                       /* Saving different screens */
   value(FNAME,STRING)                                   /* Value of a field */
   field(FNAME,TYPE,ROW,COL,LEN)                        /* Screen definition */
   txtfield(ROW,COL,LEN,STRING)
   windsize(ROW,COL).
   notopline

/* database predicates used by VSCRHND */
   windowstart(Row,Col)
   mycursord(Row,Col)

/* database declarations used in LINEINP, which SCRHND calls*/

   lineinpstate(STRING,COL)
   lineinpflag
```

Example
scrhnd(on,_)

Contained in
SCRHND.PRO (real-screen version); VSCRHND.PRO (virtual-screen version)

Environment
TDOMS.PRO, TPREDS.PRO, STATUS.PRO, LINEINP.PRO

See also
createwindow, field_action, field_value, noinput

sector

Predicate
sector(Row,Col,Rad,Incr,StartAngle,EndAngle,BorderColor,FillColor,Fill)

Declaration
Sector(VRow,VCol,VRadius,Increment,Degrees,Degrees,Color,Color,Fill)
– (i,i,i,i,i,i,i,i,i) language c

Flow pattern
Sector(i,i,i,i,i,i,i,i,i)

Function
Draws a sector of a circle.

Parameters
Draws a sector of the circle with the center at (*VRow,VCol*) with the given *Radius* and the given *Color*. The sector extends from the *StartAngle* to the *EndAngle*, measured relative to a horizontal radius pointing towards the right-hand side of the screen. The predicate can also fill the sector with the specified *FillColor*. The sector is filled according to the *Fill*-parameter (1 indicates fill, 0 indicates no-fill). When the sector is filled, *Incr* is the angle-increment used when drawing the succession of radii that fill the sector.

Remarks
The predicate *sector* is implemented in C.

Range for Rows:	0-31999
Range for Columns:	0-31999
Range for Increment:	1-360
Range for Angles:	0-360

It requires the creation of a project.

Example
sector(16000,16000,2000,1,22,59,1,1)

Contained in
GRAPHICS.OBJ

Environment
GGLOBS.PRO, GDOMS.PRO

sendBreak_RS232

Predicate
sendBreak_RS232

Declaration
sendBreak_RS232 – language c

Flow pattern
sendBreak_RS232

Function
Sends a break signal to the specified I/O port. *sendBreak_RS232* fails if the specified port has not been opened. The receiver should detect the break signal using the *status_RS232* tool predicate.

Parameters
Sends a break signal to the I/O port, according to the normal code (1 means COM1, 2 means COM2).

Remarks
Requires the definition of a project.

Example
sendBreak_RS232

Contained in
MODEM.OBJ

Environment
COMGLOBS.PRO, TICKS.OBJ

See also
status_RS232

setModemMode

Predicate
setModemMode(PortNo,CommandAtt,CommandTerminator,BreakTime)

Declaration
setModemMode(Integer, String, Char, Integer) – (i,i,i,i) language c

Flow pattern
setModemMode(i,i,i,i)

Function
Sets the communication mode for a Hayes-compatible modem.

Parameters
Sets the modem mode as follows:

PortNo	= 1 means COM1 serial communication port. = 2 means COM2 serial communication port.
CommandAtt	= Modem command prefix—normally *AT* or the empty string.
CommandTerminator	= Command suffix—normally *CR ('\13')*. = '\0' denotes 'no data terminator'.
BreakTime	= A number in the range 0 to 32,767 denoting the length of time (in hundredths of a second) for which a break signal is to be placed on the line, normally in the range 10 to 25.

Remarks
Before calling *setModemMode*, COM1 or COM2 must already have been initialized using the *openRS232* tool predicate. Indeed, *setModemMode* fails if the specified port has not been opened. Requires the definition of a project.

Example
setModemMode(2, "AT", '\013',1n0)

Contained in
MODEM.OBJ

Environment
COMGLOBS.PRO, TICKS.OBJ

See also
openRS232

setverify

Predicate
setverify(Switch)

Declaration
setverify(Integer)

Flow pattern
setverify(i)

Function
Sets the DOS verify switch. When the verify switch is on, every write operation in the DOS system is followed by a read operation to ensure that the data has been stored correctly.

Parameters
If called with *Switch* bound to zero, then the DOS verify switch is set to OFF; if *Switch* is bound to 1 before a call, then the DOS verify switch is set to ON.

Example
setverify(1)

Contained in
BIOS.PRO

See also
getverify

shiftScale

Predicate
shiftScale(MyScaleNo)

Declaration
shiftScale(ScaleNo)

Flow pattern
shiftScale(i)

Function
Used to move between scales that have been defined using *defineScale.*

Parameters
MyScaleno is the identifier for the scale returned from *defineScale.*

Example
shiftScale(2)

Contained in
GGRAPH.PRO

Environment
TDOMS.PRO, GDOMS.PRO, TPREDS.PRO, GPREDS.PRO

See also
defineScale, findScale

status_RS232

Predicate
status_RS232(PortNo,Status)

Declaration
status_RS232(Integer,Integer) – (i,o) language c

Flow pattern
status_RS232(i,o)

Function
The tool predicate *status_RS232* returns information concerning the current state of transmission. This information could be used during debugging, for example, or in the production of an error-checking and error-correcting, file-transfer package. *status_RS232* returns a status value that is a bit mask, so it is often necessary to use the **bitand** standard predicate to de-mask the value. The status value is reset before each write and read operation, and it's good practice to check the transmission status after each transmission. *status_RS232* fails if the specified I/O port has not been opened.

Parameters
If *PortNo* is bound to the code for an opened I/O port (*PortNo=1* means COM1, *PortNo=2* means COM2), *Status* is bound to the bit-mask value, representing the current tranmission status as shown below:

Status	= 0	Transmission ok.
	= 1	Input characters have been lost because the input queue was full when characters were received.
	= 2	Parity error detected.
	= 4	Over-run detected.
	= 8	Framing error detected.
	= 16	Break signal detected.
	= 32	An XOFF has been received.
	= 64	An XON has been received.
	= 128	An XOFF has been transmitted.
	= 256	An XON has been transmitted.
	= 512	Input buffer is empty when trying to read.
	= 1024	Output buffer is full when trying to write.

Remarks
Requires a project to be defined.

Example
status_RS232(1,X)

Contained in
SERIAL.OBJ

Environment
COMGLOBS.PRO, TICKS.OBJ

See also
openRS232

temp_helpcontext

Predicate
temp_helpcontext(NameOfHelpContext)

Declaration
temp_helpcontext(HelpContext)

Flow pattern
temp_helpcontext(i)

Function
If there is any possibility of the program failing while in the current context, instead of *push_helpcontext*, use the predicate *temp_helpcontext*. It automatically removes the *helpcontext* on backtracking.

Parameters
Enables the Help predicate to access the Help text known as *NameOfHelpContext* created by HELPDEF.PRO.

Remarks
HELPDEF.PRO is described in "Context-Sensitive Help," Chapter 2.

Example
temp_helpcontext(submenu2)

Contained in
HELP.PRO

Environment
TDOMS.PRO, TPREDS.PRO

See also
push_helpcontext

tempstatus

Predicate
tempstatus(StatusWindowAttribute, StringEntry)

Declaration
tempstatus(Attr,String)

Flow pattern
tempstatus(i,i)

Function
Used instead of *makestatus* when a call to *makestatus* may fail. In the event of backtracking to a *tempstatus* call, the status line is removed.

Parameters
The text to which *StringEntry* is bound is displayed in the status line, which has attribute given by *StatusWindowAttr*.

Remarks
Uses window number 83 for the status line.

Example
tempstatus(7,"Press H for Help")

Contained in
STATUS.PRO

Environment
TDOMS.PRO, TPREDS.PRO

See also
makestatus, refreshstatus, removestatus

ticks

Predicate
ticks(TimeInHundredthsOfSec)

Declaration
ticks(Integer)

Flow patterns *ticks(i)*

Function
ticks is used to suspend program execution for a specified period of time.

Parameters
Used to specify the period for the wait state in hundredths of a second. So, if *TimeInHundredthsOfSec* is bound to 150, program execution is suspended for 1.5 seconds. Valid values range from 1 and 32767.

Remarks
Requires the definition of a project.

Contained in
TICKS.OBJ

Environment
COMGLOBS.PRO

treemenu

Predicate
treemenu(DirectionOfTreeDisplay, MyTree, Choice)

Declaration
treemenu(Symbol, Tree, Selector)

where TREE and SELECTOR are defined as follows:

```
SELECTOR = Integer
TREE = tree(String, SELECTOR, TREELIST)
TREELIST = TREE*
```

(In actual use, the SELECTOR can be of any TYPE convenient to the programmer).

Flow pattern
treemenu(i,i,o)

Function
Implements a tree menu on a virtual screen. (The cursor keys can be used to move around the tree.) The user makes a menu selection by pressing Return or F10.

Parameters
Displays a tree menu in which the tree is drawn *up, down, left,* or *right* according to the value *DirectionOfTreeDisplay* is bound to.

MyTree must be bound to a TREE definition that determines, for each item, the text describing that item, its position in the tree, and the code to be returned when that item is selected. To this end, use is made of the *tree* functor, which takes three arguments. The first is a String that gives the text for that menu item; the second is the code to be returned if that menu item is selected; and the third is a Treelist that specifies the rest of the tree.

Choice becomes bound to the code for the menu item chosen.

Example
```
treemenu(up,tree("start",1,
        [tree("start_problem",2,[tree("gasoline_help",3,[])],)]),
        CH)
```

Contained in
TREEMENU.PRO

Environment
TDOMS.PRO, TPREDS.PRO

txCh_RS232

Predicate
txCh_RS232(PortNo,CH)

Declaration
txCh_RS232(Integer,Char) – (i,i) language c

Flow pattern
txCh_RS232(i,i)

Function
txCh_RS232 places a character in the output buffer if the buffer is not full. That character is then transmitted when the receiver is ready. *txCh_RS232* fails if the output buffer is full or the specified communication port is not open.

Parameters
The character *CH* is transmitted to the output buffer for port number *PortNo,* where *PortNo=1* means the COM1 serial-communication port and *PortNo*=2 means the COM2 serial-communication port.

Remarks
Since the low-level transmission is interrupt driven, it is fully transparent when viewed from a Turbo Prolog program: The program will not normally be aware of when transmission from the buffer takes place. However, the status of a transmission can always be monitored by calling the predicate *status_RS232.*

Requires definition of a project.

Example
txCh_RS232(1, CH)

Contained in
SERIAL.OBJ

Environment
COMGLOBS.PRO

See also
status_RS232

txStr_Modem

Predicate
txStr_Modem(txString,NoOfCharsTransmitted)

Declaration
txStr_Modem(String, Integer) – (i,o) language c

Flow pattern
txStr_Modem(i,o)

Function
Sends a command or pure data string to the modem using the parameters set by the most recent call to the *SetModemMode* predicate; The string is sent via the serial I/O port affected by that call. *txStr_Modem* fails if the modem port is not initialized.

Parameters
With *txString* bound to the command or data string to be transmitted, the predicate binds *NoOfCharsTransmitted* to the actual number of characters transmitted. (This may be different from the number of characters in *txString*.) Note that the length of the string transmitted can never be larger than the size of the output buffer.

Remarks
Requires the definition of a project.

Example
txStr_Modem("Z",_)

Contained in
MODEM.OBJ

Environment
COMGLOBS.PRO

See also
setModemMode

virtual_text

Predicate
virtual_text(MyVirtualRow,MyVirtualCol,TextRow,TextCol)

Declaration
virtual_text(VRow,VCol,Row,Col)

Flow pattern
virtual_text(i,i,o,o), (o,o,i,i)

Function
Conversion between virtual coordinates and text coordinates in a graphics window.

Parameters
(*MyVirtualRow,MyVirtualCol*) are the virtual coordinates; (*TextRow,TextCol*) are the text coordinates.

Example
virtual_text(R,C,24,79)

Contained in
GGRAPH.PRO

Environment
TDOMS.PRO, GDOMS.PRO, TPREDS.PRO, GPREDS.PRO

writescr (programmer defined)

Predicate
writescr

Declaration
writescr

Flow pattern
None—takes no parameters.

Function
Restores the display after window resizing.

Parameters
None

Remarks
writescr is called by *resizewindow* so that you can refresh the display after a user has resized a window. For example, it is used in SCRDEF.PRO when the user resizes the screen window.

Contained in
RESIZE.PRO

Environment
TDOMS.PRO, TPREDS.PRO

See also
resizewindow

A P P E N D I X

A

Compiling a Project

The section "Modular Programming" in Chapter 11 of the *Turbo Prolog Owner's Handbook* explains this innovative Turbo Prolog feature. However, a brief introduction follows.

Program modules are written, edited, and compiled separately, then linked to create a single executable program. To make a change in the program, you edit and recompile only the relevant module. In addition, since all predicate and domain names are local, different modules can use the same name for diverse functions.

Two concepts manage modular programming in Turbo Prolog: *projects* and *global declarations*. Projects keep track of the modules that make up a program, while global declarations make it possible to perform type checking and to call predicates across module boundaries.

The first step in modular programming is to name the project and create a corresponding librarian file (a .LIB file in the Module List) containing the names of the project's modules. Then you list all global declarations in a single file, which you can include in each pertinent module via an **include** directive. Finally, you compile and link the modules. All of these steps can be accomplished using menu options in the Turbo Prolog menu.

Compiling Projects Provided in the Toolbox

To compile example programs that require .OBJ files, select Options from the Turbo Prolog main menu. Next select Project from the pull-down menu, and enter the project name at the prompt.

The Prolog file XBAR.PRO, for example, also has the project file XBAR.PRJ on the disk or directory containing the sample programs. XBAR.PRJ lists the files that Turbo Prolog is to compile and link together to create the XBAR.EXE file.

If you're not sure of a project name, press Return and a window containing a directory of all project files pops up. You can use the arrow keys to select the appropriate project file. Once you have selected a project, select Compile on the main menu. All Prolog files are automatically compiled and linked with the .OBJ files. (**NOTE:** The project selected will be compiled, not the file in the editor. Turbo Prolog will prompt you to save the file in the editor, if you need to.)

Creating Your Own Project File

To create your own project file, select Files from the main menu and Module List from the pull-down menu. Then, in the library editor (Turbo Prolog editing window), enter the names of the Prolog files and the .OBJ files with a + extension (not the regular .PRO or .OBJ extensions). For example, the XPIE.PRJ file contains

```
graphics+
xpie+
```

where *graphics*+ is the tool file GRAPHICS.OBJ, and *xpie*+ is the example program XPIE.PRO.

Glossary

arguments The objects and variables in a relation; in Turbo Prolog, the arguments are contained within parentheses.

associative (left/right) Refers to the order in which operations are performed. Turbo Prolog operators are left associative by default; the keyword *rightassoc* must precede operations that are right associative.

attribute A positive whole number that determines the characteristics of a window's display, such as color, normal/inverse video, and blinking/non-blinking.

backtracking A Turbo Prolog built-in mechanism that, when the evaluation of a sub-goal is complete, returns to the previous sub-goal and tries to satisfy it in a different way.

Backus Naur Form (BNF) A meta-language or system of notation used to specify or describe the syntax of a language.

bootstrapping Using part of a computer program to bring about another version of the program.

call a predicate Bringing a subroutine into effect by jumping to the specified entry point. This transfers control of execution to the subprogram; after execution ends, the main program resumes with the statement after the call.

clip To use only that part of an image that fits inside the specified active window.

dialog window The window in which external goals are given and their results recorded.

expert system A program that has been fed enough information for it to mimic the ability of an expert in a specified field. Its database is supplemented by an inference engine so that it can analyze given facts.

expression A set of symbols that can be evaluated for a value, or a notation representing a value.

external goal A goal entered in the dialog window by the user and given to the program currently being worked on.

fact A relation between objects.

global A qualifier that allows more than one module access to certain domains and predicates.

goal The collection of sub-goals that Turbo Prolog attempts to satisfy.

grammar A description of the specifications and structures that are allowed in a source text.

identifier A symbolic name you define to represent variables or constants in a program.

integer A whole number in the range –32768 to 32767.

module A Turbo Prolog program with global declarations that is part of a project file. You can write, edit, and compile modules separately, and then combine them into a single executable program.

non-terminals Special symbols that denote sets of strings in a grammar.

parser A routine or program that analyzes the syntactical structure of a string of characters, based on the syntax specified by the programming language.

parameters The objects and variables in a relation.

PC Personal computer; usually refers to an IBM PC or compatible.

pixel Literally, picture element. The smallest element of a display that can be assigned color and intensity.

poll To interrogate transmission devices to determine readiness to send or receive data.

predicate The specification of the name of the relation involved in a fact or rule, and the types of objects in the relation.

production A translation rule that you supply for forming valid sentences in the language specified by the grammar.

production rule Specifications that define the ways in which grammatical structures can be built from one another and from terminals.

project A Turbo Prolog program consisting of more than one module.

rule A relationship between a fact and a list of sub-goals that must be satisfied for the fact to be true.

sentence A list of terminal symbols.

start symbol Where a parser begins determining how to parse its source input; it and denotes the language being defined.

statement A set of instructions.

string An arbitrary number of characters enclosed by a pair of double quotation marks (" ").

terminals The basic symbols from which sentences are created in a language.

token A name, an unsigned (real or integer) number, or a non-space character.

variable A name that represents the (possibly unknown) value of an object.

Index

Borland Software

BORLAND
INTERNATIONAL

4585 Scotts Valley Drive, Scotts Valley, CA 95066

Available at better dealers nationwide.
To order by credit card, call (800) 255-8008; CA (800) 742-1133;
CANADA (800) 237-1136.

SIDEKICK:® THE DESKTOP ORGANIZER

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 to jot down notes and edit files up to 25 pages long.

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

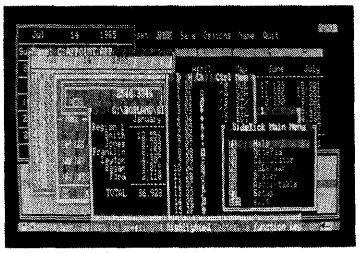
An Autodialer for all your phone calls. It will look up and dial telephone numbers for you. (A modem is required to use this function.)

A Monthly Calendar from 1901 through 2099.

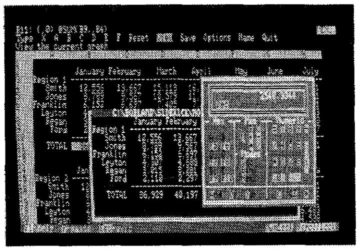
Appointment Calendar to remind you of important meetings and appointments.

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, Appointment Calendar, 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."
—Gary 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

Suggested Retail Price: $84.95 (not copy protected)

Minimum system configuration: IBM PC, XT, AT, PCjr and true compatibles. PC-DOS (MS-DOS) 2.0 or greater. 128K RAM. One disk drive. A Hayes-compatible modem, IBM PCjr internal modem, or AT&T Modem 4000 is required for the autodialer function.

BORLAND INTERNATIONAL

Traveling SIDEKICK®

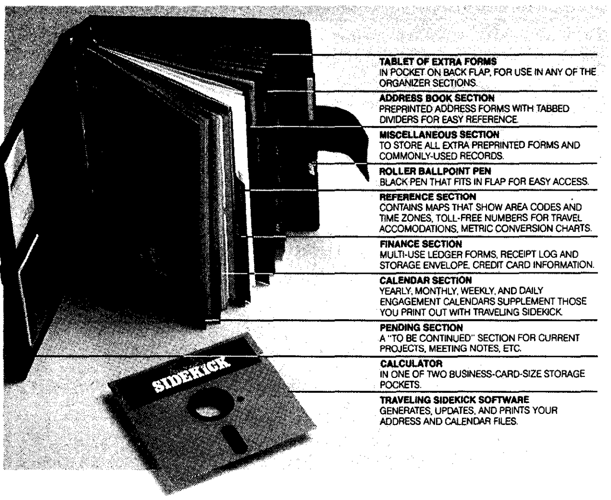
The Organizer For The Computer Age!

Traveling SideKick is *BinderWare,*® both a binder you take with you when you travel and a software program—which includes a Report Generator—that *generates* and *prints out* all the information you'll need to take with you.

Information like your phone list, your client list, your address book, your calendar, and your appointments. The appointment or calendar files you're already using in your SideKick® can automatically be used by your Traveling SideKick. You don't waste time and effort reentering information that's already there.

One keystroke prints out a form like your address book. No need to change printer paper; you simply punch three holes, fold and clip the form into your Traveling SideKick binder, and you're on your way. Because Traveling SideKick is CAD (Computer-Age Designed), you don't fool around with low-tech tools like scissors, tape, or staples. And because Traveling SideKick is electronic, it works this year, next year, and all the "next years" after that. Old-fashioned daytime organizers are history in 365 days.

What's inside Traveling SideKick



TABLET OF EXTRA FORMS
IN POCKET ON BACK FLAP, FOR USE IN ANY OF THE ORGANIZER SECTIONS.
ADDRESS BOOK SECTION
PREPRINTED ADDRESS FORMS WITH TABBED DIVIDERS FOR EASY REFERENCE.
MISCELLANEOUS SECTION
TO STORE ALL EXTRA PREPRINTED FORMS AND COMMONLY-USED RECORDS.
ROLLER BALLPOINT PEN
BLACK PEN THAT FITS IN FLAP FOR EASY ACCESS.
REFERENCE SECTION
CONTAINS MAPS THAT SHOW AREA CODES AND TIME ZONES, TOLL-FREE NUMBERS FOR TRAVEL ACCOMODATIONS, METRIC CONVERSION CHARTS.
FINANCE SECTION
MULTI-USE LEDGER FORMS, RECEIPT LOG AND STORAGE ENVELOPE, CREDIT CARD INFORMATION.
CALENDAR SECTION
YEARLY, MONTHLY, WEEKLY, AND DAILY ENGAGEMENT CALENDARS SUPPLEMENT THOSE YOU PRINT OUT WITH TRAVELING SIDEKICK.
PENDING SECTION
A "TO BE CONTINUED" SECTION FOR CURRENT PROJECTS, MEETING NOTES, ETC.
CALCULATOR
IN ONE OF TWO BUSINESS-CARD-SIZE STORAGE POCKETS.
TRAVELING SIDEKICK SOFTWARE
GENERATES, UPDATES, AND PRINTS YOUR ADDRESS AND CALENDAR FILES.
SIDEKICK

What the software program and its Report Generator do for you before you go—and when you get back

Before you go:

- Prints out your calendar, appointments, addresses, phone directory, and whatever other information you need from your data files

When you return:

- Lets you quickly and easily enter all the new names you obtained while you were away into your SideKick data files

It can also:

- Sort your address book by contact, zip code or company name
- Print mailing labels
- Print information selectively
- Search files for existing addresses or calendar engagements

Suggested Retail Price: $69.95 (not copy protected)

Minimum system configuration: IBM PC, XT, AT, Portable, PCjr, 3270 and true compatibles. PC-DOS (MS-DOS) 2.0 or later. 256K RAM mimimum.

BORLAND
INTERNATIONAL

BOR 0083A

SUPERKEY®: THE PRODUCTIVITY BOOSTER

RAM-resident
Increased productivity for IBM®PCs or compatibles

SuperKey's simple macros are electronic shortcuts to success. By letting you reduce a lengthy paragraph into a single keystroke of your choice, SuperKey eliminates repetition.

SuperKey turns 1,000 keystrokes into 1!
SuperKey can record lengthy keystroke sequences and play them back at the touch of a single key. Instantly. Like magic.

In fact, with SuperKey's simple macros, you can turn "Dear Customer: Thank you for your inquiry. We are pleased to let you know that shipment will be made within 24 hours. Sincerely," into the one keystroke of your choice!

SuperKey keeps your confidential files—confidential!
Without encryption, your files are open secrets. Anyone can walk up to your PC and read your confidential files (tax returns, business plans, customer lists, personal letters, etc.).

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 correctly. SuperKey also implements the U.S. government Data Encryption Standard (DES).

- ☑ RAM resident—accepts new macro files even while running other programs
- ☑ Pull-down menus
- ☑ Superfast file encryption
- ☑ Choice of two encryption schemes
- ☑ On-line context-sensitive help
- ☑ One-finger mode reduces key commands to single keystroke
- ☑ Screen OFF/ON blanks out and restores screen to protect against "burn in"
- ☑ Partial or complete reorganization of keyboard
- ☑ Keyboard buffer increases 16 character keyboard "type-ahead" buffer to 128 characters
- ☑ Real-time delay causes macro playback to pause for specified interval
- ☑ Transparent display macros allow creation of menus on top of application programs
- ☑ Data entry and format control using "fixed" or "variable" fields
- ☑ Command stack recalls last 256 characters entered

Suggested Retail Price: $99.95 (not copy protected)

Minimum system configuration: IBM PC, XT, AT, PCjr, and true compatibles. PC-DOS (MS-DOS) 2.0 or greater. 128K RAM. One disk drive.

BORLAND
INTERNATIONAL

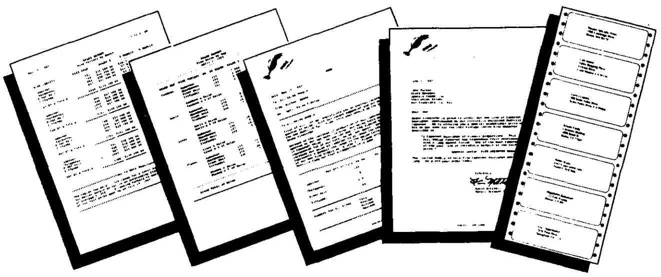
REFLEX:® THE DATABASE MANAGER

The high-performance database manager that's so advanced it's easy to use!

Lets you organize, analyze and report information faster than ever before! If you manage mailing lists, customer files, or even your company's budgets—Reflex is the database manager for you!

Reflex is the acclaimed, high-performance database manager you've been waiting for. Reflex extends database management with business graphics. Because a picture is often worth a 1000 words, Reflex lets you extract critical information buried in mountains of data. With Reflex, when you look, you see.

The ***REPORT VIEW*** allows you to generate everything from mailing labels to sophisticated reports. You can use database files created with Reflex or transferred from Lotus 1-2-3,® dBASE,® PFS: File,® and other applications.



Reflex: The Critics' Choice

". . . if you use a PC, you should know about Reflex . . . may be the best bargain in software today."
Jerry Pournelle, BYTE

"Everyone agrees that Reflex is the best-looking database they've ever seen."
Adam B. Green, InfoWorld

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

Reflex: don't use your PC without it!
Join hundreds of thousands of enthusiastic Reflex users and experience the power and ease of use of Borland's award-winning Reflex.

Suggested Retail Price: $149.95 (not copy protected)

Minimum system configuration: IBM PC, XT, AT, and true compatibles. 384K RAM minimum. IBM Color Graphics Adapter, Hercules Monochrome Graphics CArd, or equivalent. PC-DOS (MS-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®

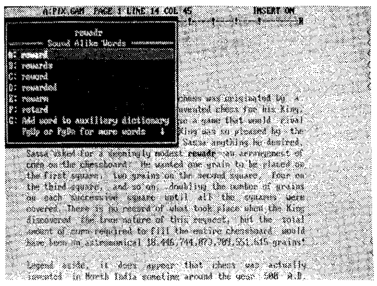
Turbo Lightning teams up with the Random House Concise Word List to check your spelling as you type!

Turbo Lightning, using the 80,000-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 one key and the misspelled word is instantly replaced with the correct word.

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

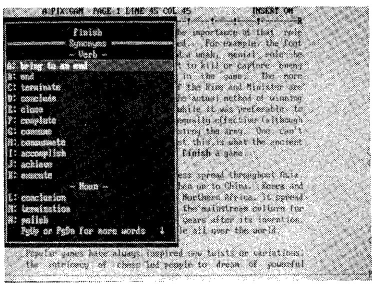
Turbo Lightning lets you choose just the right word from a list of alternates, so you don't say the same thing the same way every time. Once Turbo Lightning opens the Thesaurus window, you see a list of alternate words; select the word you want, press ENTER and your new word will instantly replace the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

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



```
A:FIX.GAM  PAGE 1 LINE 14 COL 45            INSERT ON
                reuadr
   ------ Sound Alike Words ------
A: reward
B: rewards
C: reword
D: rewarded
E: rewarm
F: retard
G: Add word to auxiliary dictionary
   PgUp or PgDn for more words   ↓
                             chess was originated by a
                             invented chess for his King,
                             se a game that would rival
                             King was so pleased by the
                             Sassa anything he desired.
Sassa asked for a seemingly modest reuadr  an arrangement of
corn on the chessboard.  He wanted one grain to be placed on
the first square,  two grains on the second square,  four on
the third square,  and so on,  doubling the number of grains
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 grains!

Legend aside,  it does appear that chess was actually
invented  in North India sometime around the year 500 A.D.
```

The Turbo Lightning Proofreader



```
A:FIX.GAM  PAGE 1 LINE 45 COL 45            INSERT ON
                finish
   ------ Synonyms ------
                - Verb -
A: bring to an end
B: end
C: terminate
D: conclude
E: close
F: complete
G: consume
H: consummate
I: accomplish
J: achieve
K: execute
                - Noun -
L: conclusion
M: termination
N: polish
   PgUp or PgDn for more words   ↓
                             he importance of that role
                             ed.  For example, the foot
                             d a weak, menial role he
                             t to kill or capture enemy
                             in the game.  The more
                             f the King and Minister are
                             he actual method of winning
                             while it was preferable to
                             equally effective (although
                             stroy the army.  One can't
                             at this is what the ancient
                             finish a game.

                             ess spread throughout Asia
                             hen on to China,  Korea and
                             Northern Africa. It spread
                             the mainstream culture for
                             years after its invention,
                             le all over the world.
Popular games have always inspired new twists or variations;
the intricacy of chess led people to dream of powerful
```

The Turbo Lightning Thesaurus

You can teach Turbo Lightning new words

You can *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 Concise Word List 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.

Suggested Retail Price: $99.95 (not copy protected)

Minimum system configuration: IBM PC, XT, AT, PCjr, and true compatibles with 2 floppy disk drives. PC-DOS (MS-DOS) 2.0 or greater. 256K RAM. Hard disk recommended.

BORLAND INTERNATIONAL

Your Development Toolbox and Technical Reference Manual for Turbo Lightning®

L I G H T N I N G WORDWIZARD™

Lightning Word Wizard includes complete, commented Turbo Pascal® source code and all the technical information you'll need to understand and work with Turbo Lightning's "engine." More than 20 fully documented Turbo Pascal procedures reveal powerful Turbo Lightning engine calls. Harness the full power of the complete and authoritative Random House® Concise Word List and Random House Thesaurus.

Turbo Lightning's "Reference Manual"
Developers can use the versatile on-line examples to harness Turbo Lightning's power to do rapid word searches. Lightning Word Wizard is the forerunner of the database access systems that will incorporate and engineer the Turbo Lightning Library® of electronic reference works.

The ultimate collection of word games and crossword solvers!
The excitement, challenge, competition, and education of four games and three solver utilities—puzzles, scrambles, spell-searches, synonym-seekings, hidden words, crossword solutions, and more. You and your friends (up to four people total) can set the difficulty level and contest the high-speed smarts of Lightning Word Wizard!

Turbo Lightning—Critics' Choice

"Lightning's good enough to make programmers and users cheer, executives of other software companies weep." **Jim Seymour, *PC Week***

"The real future of Lightning clearly lies not with the spelling checker and thesaurus currently included, but with other uses of its powerful look-up engine." **Ted Silveira, *Profiles***

"This newest product from Borland has it all." **Don Roy, *Computing Now!***

Minimum system configuration: IBM PC, XT, AT, PCjr, Portable, and true compatibles. 256K RAM minimum. PC-DOS (MS-DOS) 2.0 or greater. Turbo Lightning software required. Optional—Turbo Pascal 3.0 or greater to edit and compile Turbo Pascal source code.

BORLAND INTERNATIONAL

Suggested Retail Price: $69.95 (not copy protected)

BOR0087B

TURBOPASCAL®

Version 3.0 with 8087 support and BCD reals

Free MicroCalc Spreadsheet With Commented Source Code!

FEATURES:

One-Step Compile: No hunting & fishing expeditions! Turbo finds the errors, takes you to them, lets you correct them, and 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.

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

LOOK AT TURBO NOW!

- ✓ More than 500,000 users worldwide.
- ✓ Turbo Pascal is the de facto industry standard.
- ✓ Turbo Pascal wins 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.

Suggested Retail Price: $99.95; CP/M®-80 version without 8087 and BCD: $69.95

Features 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.

Minimum system configuration: 128K RAM minimum. Includes 8087 & BCD features for 16-bit MS-DOS 2.0 or later and CP/M-86 1.1 or later. CP/M-80 version 2.2 or later 48K RAM minimum (8087 and BCD features not available). 8087 version requires 8087 or 80287 co-processor.

BORLAND
INTERNATIONAL

TURBO PASCAL

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 database applications. It's another set of tools from Borland that will give even the beginning programmer the expert's edge.

THE TOOLS YOU NEED!

TURBO ACCESS 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.

TURBO SORT: The fastest way to sort data using the QUICKSORT algorithm—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 diskette 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.

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 Pournell, BYTE MAGAZINE***

"The Turbo Database Toolbox is solid enough and useful enough to come recommended."
—Jeff Duntemann, PC TECH JOURNAL

Suggested Retail Price: $69.95 (not copy protected)

Minimum system configuration: 128K RAM and one disk drive (CP/M-80: 48K). 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 1.0 or greater. 8-bit systems: Turbo Pascal 2.0 or greater for CP/M-80 2.2 or greater.

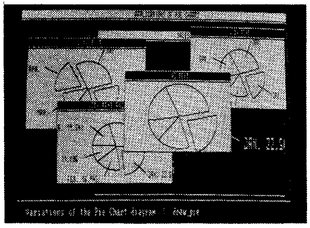
BORLAND
INTERNATIONAL

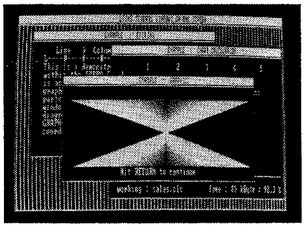
TURBO PASCAL GRAPHIX TOOLBOX®

A Library of Graphics Routines for Use with Turbo Pascal®

High-resolution graphics for your IBM™ PC, AT,™ XT,™ PCjr™, true PC compatibles, and the Heath Zenith Z-100.™ Comes complete with graphics window management.

Even if you're new to Turbo Pascal programming, the Turbo Pascal Graphix Toolbox will get you started right away. It's a collection of tools that will get you right into the fascinating world of high-resolution business graphics, including graphics window management. You get immediate, satisfying results. And we keep Royalty out of American business because you don't pay any—even if you distribute your own compiled programs that include all or part of the Turbo Pascal Graphix Toolbox procedures.





What you get includes:

- Complete commented source code on disk.
- Tools for drawing simple graphics.
- Tools for drawing complex graphics, including curves with optional smoothing.
- Routines that let you store and restore graphic images to and from disk.
- Tools allowing you to send screen images to Epson®-compatible printers.
- Full graphics window management.
- Two different font styles for graphic labeling.
- Choice of line-drawing styles.
- Routines that will let you quickly plot functions and model experimental data.
- And much, much more . . .

"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."

John Markov & Paul Freiberger, syndicated columnists.

If you ever plan to create Turbo Pascal programs that make use of business graphics or scientific graphics, you need the Turbo Pascal Graphix Toolbox.

Suggested Retail Price: $69.95 (not copy protected)

Minimum system configuration: IBM PC, XT, AT, PCjr, true compatibles and the Heath Zenith Z-100. Turbo Pascal 3.0 or later. 192K RAM minimum. Two disk drives and an IBM Color Graphics Adapter (CGA), IBM Enhanced Graphics Adapter (EGA), Hercules Graphics Card or compatible.

BORLAND
INTERNATIONAL

TURBO PASCAL 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.

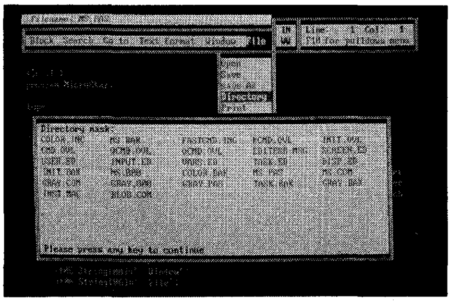
Create your own word processor. We provide all the editing routines. You plug in the features you want. You could build a WordStar®-like editor with pull-down menus like Microsoft's® Word, and make it work as fast as WordPerfect.®

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:

- Wordwrap
- UN-delete last line
- 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.



```
Directory mask:
Please press any key to continue
```

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.

☑ ***Multitasking.*** 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.**

Suggested Retail Price: $69.95 (not copy protected)

Minimum system configuration: IBM PC, XT, AT, 3270, PCjr, and true compatibles. PC-DOS (MS-DOS) 2.0 or greater. 192K RAM. You must be using Turbo Pascal 3.0 for IBM and compatibles.

BORLAND
INTERNATIONAL

BOR 0067B

TURBO PROLOG™

the natural language of Artificial Intelligence

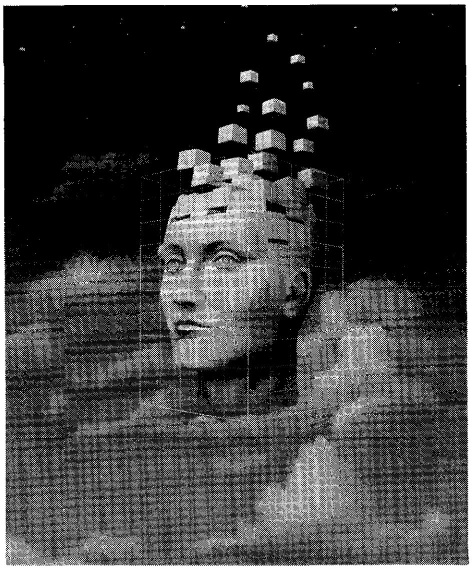
STEP-BY-STEP TUTORIAL AND DEMO PROGRAMS WITH SOURCE CODE INCLUDED!

Turbo Prolog brings fifth-generation supercomputer power to your IBM® PC!

Turbo Prolog takes programming into a new, natural, and logical environment

With Turbo Prolog, because of its natural, logical approach, both people new to programming *and* professional programmers can build powerful applications such as expert systems, customized knowledge bases, natural language interfaces, and smart information management systems.

Turbo Prolog is a *declarative* language which uses deductive reasoning to solve programming problems.



Turbo Prolog provides a fully integrated programming environment like Borland's Turbo Pascal,® the *de facto* worldwide standard.

You get the complete Turbo Prolog programming system

You get the 200-page manual you're holding, software that includes the lightning-fast Turbo Prolog six-pass compiler and interactive editor, and the free GeoBase natural query language database, which includes commented source code on disk, ready to compile. (GeoBase is a complete database designed and developed around U.S. geography. You can modify it or use it "as is.")

Turbo Prolog's development system includes:

- ☐ A complete Prolog compiler that is a variation of the Clocksin and Mellish Edinburgh standard Prolog.
- ☐ A full-screen interactive editor.
- ☐ Support for both graphic and text windows.
- ☐ All the tools that let you build your own expert systems and **AI** applications with unprecedented ease.

Minimum system configuration: IBM PC, XT, AT, Portable, 3270, PCjr and true compatibles. PC-DOS (MS-DOS) 2.0 or later. 384K RAM minimum.

Suggested Retail Price: $99.95 (not copy protected)

BORLAND INTERNATIONAL

TURBO C®

Includes free MicroCalc spreadsheet with source code

A complete interactive development environment

With Turbo C, you can expect what only Borland delivers: Quality, Speed, Power and Price. And with its compilation speed of more than 7000 lines a minute, Turbo C makes everything else look like an exercise in slow motion.

Turbo C: The C compiler for both amateurs and professionals
If you're just beginning and you've "kinda wanted to learn C," now's your chance to do it the easy way. Turbo C's got everything to get you going. If you're already programming in C, switching to Turbo C will considerably increase your productivity and help make your programs both smaller and faster.

Turbo C: a complete interactive development environment
Like Turbo Pascal® and Turbo Prolog,™ Turbo C comes with an interactive editor that will show you syntax errors right in your source code. Developing, debugging, and running a Turbo C program is a snap!

Technical Specifications

- ✓ Compiler: One-pass compiler generating native in-line code, linkable object modules and assembler. The object module format is compatible with the PC-DOS linker. Supports small, medium, compact, large, and huge memory model libraries. Can mix models with near and far pointers. Includes floating point emulator (utilizes 8087/80287 if installed).
- ✓ Interactive Editor: The system includes a powerful, interactive full-screen text editor. If the compiler detects an error, the editor automatically positions the cursor appropriately in the source code.
- ✓ Development Environment: A powerful "Make" is included so that managing Turbo C program development is easy. Borland's fast "Turbo Linker" is also included. Also includes pull-down menus and windows. Can run from the environment or generate an executable file.
- ✓ Links with relocatable object modules created using Borland's Turbo Prolog into a single program.
- ✓ ANSI C compatible.
- ✓ Start-up routine source code included.
- ✓ Both command line and integrated environment versions included.

"Sieve" benchmark (25 iterations)

	Turbo C	*Microsoft® C*	*Lattice C*
Compile time	***3.89***	*16.37*	*13.90*
Compile and link time	***9.94***	*29.06*	*27.79*
Execution time	***5.77***	*9.51*	*13.79*
Object code size	***274***	*297*	*301*
Price	***$99.95***	*$450.00*	*$500.00*

Benchmark run on a 6 Mhz IBM AT using Turbo C version 1.0 and the Turbo Linker version 1.0; Microsoft C version 4.0 and the MS overlay linker version 3.51; Lattice C version 3.1 and the MS object linker version 3.05.

Suggested Retail Price: $99.95* (not copy protected) *Introductory offer good through July 1, 1987.

Minimum system configuration: IBM PC, XT, AT and true compatibles. PC-DOS (MS-DOS) 2.0 or later. One floppy drive. 320K.

BORLAND INTERNATIONAL

BOR 0243

EUREKA: THE SOLVER™

The solution to your most complex equations—in seconds!

If you're a scientist, engineer, financial analyst, student, teacher, or any other professional working with equations, Eureka: The Solver can do your Algebra, Trigonometry and Calculus problems in a snap.

Eureka also handles maximization and minimization problems, plots functions, generates reports, and saves an incredible amount of time. Even if you're not a computer specialist, Eureka can help you solve your real-world mathematical problems fast, without having to learn numerical approximation techniques. Using Borland's famous pull-down menu design and context-sensitive help screens, Eureka is easy to learn and easy to use—as simple as a hand-held calculator.

X + exp(X) = 10 solved instantly instead of eventually!
Imagine you have to "solve for X," where X + exp(X) = 10, and you don't have Eureka: The Solver. What you do have is a problem, because it's going to take a lot of time guessing at "X." With Eureka, there's no guessing, no dancing in the dark—you get the right answer, right now. (PS: X = 2.0705799, and Eureka solved that one in .4 of a second!)

How to use Eureka: The Solver

It's easy.
1. Enter your equation into the full-screen editor
2. Select the "Solve" command
3. Look at the answer
4. You're done

You can then tell Eureka to
- Evaluate your solution
- Plot a graph
- Generate a report, then send the output to your printer, disk file or screen
- Or all of the above

Some of Eureka's key features

You can key in:
- ✓ A formula or formulas
- ✓ A series of equations—and solve for all variables
- ✓ Constraints (like X has to be < or = 2)
- ✓ A function to plot
- ✓ Unit conversions
- ✓ Maximization and minimization problems
- ✓ Interest Rate/Present Value calculations
- ✓ Variables we call "What happens?," like "What happens if I change this variable to 21 and that variable to 27?"

Eureka: The Solver includes

- ✓ A full-screen editor
- ✓ Pull-down menus
- ✓ Context-sensitive Help
- ✓ On-screen calculator
- ✓ Automatic 8087 math co-processor chip support
- ✓ Powerful financial functions
- ✓ Built-in and user-defined math and financial functions
- ✓ Ability to generate reports complete with plots and lists
- ✓ Polynomial finder
- ✓ Inequality solutions

Minimum system configuration: IBM PC, AT, XT, Portable, 3270 and true compatibles. PC-DOS (MS-DOS) 2.0 and later. 384K.

Suggested Retail Price: $99.95*
(not copy protected)

BORLAND
INTERNATIONAL

*Introductory price expires July 1, 1987

SIDEKICK®: THE DESKTOP ORGANIZER Release 2.0

Macintosh™

The most complete and comprehensive collection of desk accessories available for your Macintosh!

Thousands of users already know that SideKick is the best collection of desk accessories available for the Macintosh. With our new Release 2.0, the best just got better.

We've just added two powerful high-performance tools to SideKick—Outlook™: The Outliner and MacPlan™: The Spreadsheet. They work in perfect harmony with each other and *while* you run other programs!

Outlook: The Outliner

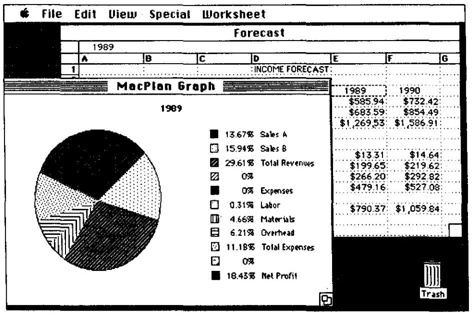
- It's the desk accessory with more power than a stand-alone outliner
- A great desktop publishing tool, Outlook lets you incorporate both text and graphics into your outlines
- Works hand-in-hand with MacPlan
- Allows you to work on several outlines at the same time

MacPlan: The Spreadsheet

- Integrates spreadsheets and graphs
- Does both formulas and straight numbers
- Graph types include bar charts, stacked bar charts, pie charts and line graphs
- Includes 12 example templates free!
- Pastes graphics and data right into Outlook creating professional memos and reports, complete with headers and footers.

SideKick: The Desktop Organizer, Release 2.0 now includes

- ☑ Outlook: The Outliner
- ☑ MacPlan: The Spreadsheet
- ☑ Mini word processor
- ☑ Calendar
- ☑ PhoneLog
- ☑ Analog clock
- ☑ Alarm system
- ☑ Calculator
- ☑ Report generator
- ☑ Telecommunications (new version now supports XModem file transfer protocol)



■	13.67%	Sales A
▨	15.94%	Sales B
▨	29.61%	Total Revenues
▨	0%	
■	0%	Expenses
□	0.31%	Labor
▥	4.66%	Materials
▤	6.21%	Overhead
▨	11.18%	Total Expenses
▨	0%	
■	18.43%	Net Profit

1989	1990
$585.94	$732.42
$683.59	$854.49
$1,269.53	$1,586.91
$13.31	$14.64
$199.65	$219.62
$266.20	$292.82
$479.16	$527.08
$790.37	$1,059.84

MacPlan does both spreadsheets and business graphs. Paste them into your Outlook files and generate professional reports.

Suggested Retail Price: $99.95 (not copy protected)

Minimum system configurations: Macintosh 512K or Macintosh Plus with one disk drive. One 800K or two 400K drives are recommended. With one 400K drive, a limited number of desk accessories will be installable per disk.

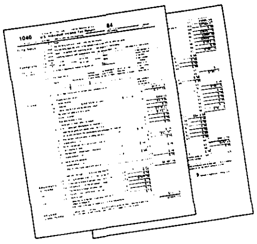
BORLAND
INTERNATIONAL

BOR 0069D

REFLEX®: THE DATABASE MANAGER

The easy-to-use relational database that thinks like a spreadsheet. Reflex for the Mac lets you crunch numbers by entering formulas and link databases by drawing on-screen lines.

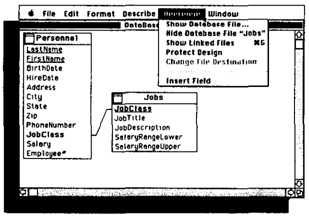
5 free ready-to-use templates are included on the examples disk:



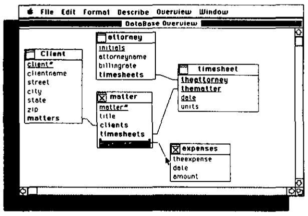
- A sample 1040 tax application with Schedule A, Schedule B, and Schedule D, each contained in a separate report document.
- A portfolio analysis application with linked databases of stock purchases, sales, and dividend payments.
- A checkbook application.
- A client billing application set up for a law office, but easily customized by any professional who bills time.
- A parts explosion application that breaks down an object into its component parts for cost analysis.

Reflex for the Mac accomplishes all of these tasks without programming—using spreadsheet-like formulas. Some other Reflex for the Mac features are:

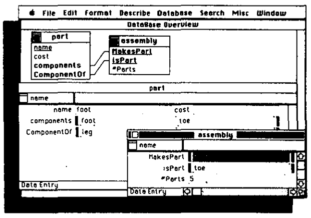
- Visual database design.
- "What you see is what you get" report and form layout with pictures.
- Automatic restructuring of database files when data types are changed, or fields are added and deleted.
- Display formats which include General, Decimal, Scientific, Dollars, Percent.
- Data types which include variable length text, number, integer, automatically incremented sequence number, date, time, and logical.
- Up to 255 fields per record.
- Up to 16 files simultaneously open.
- Up to 16 Mac fonts and styles are selectable for individual fields and labels.



After opening the "Overview" window, you draw link lines between databases directly onto your Macintosh screen.



The link lines you draw establish both visual and electronic relationships between your databases.



You can have multiple windows open simultaneously to view all members of a linked set—which are interactive and truly relational.

Critic's Choice

". . . a powerful relational database . . . uses a visual approach to information management." ***InfoWorld***

". . . gives you a lot of freedom in report design; you can even import graphics." ***A+ Magazine***

". . . bridges the gap between the pretty programs and the power programs." **Stewart Alsop, *PC Letter***

BORLAND
INTERNATIONAL

Suggested Retail Price: $99.95*
(not copy protected)

Minimum system configuration: *Macintosh 512K or Macintosh Plus with one disk drive. Second external drive recommended.*

BOR0149A

*Introductory price expires July 1, 1987

TURBO PASCAL® MACINTOSH™

The ultimate Pascal development environment

Borland's new Turbo Pascal for the Mac is so incredibly fast that it can compile 1,420 lines of source code in the 7.1 seconds it took you to read this!

And reading the rest of this takes about *5 minutes*, which is plenty of time for Turbo Pascal for the Mac to compile at least *60,000 more lines* of source code!

Turbo Pascal for the Mac does both Windows and "Units"

The *separate* compilation of routines offered by Turbo Pascal for the Mac creates modules called "Units," which can be linked to any Turbo Pascal program. This "modular pathway" gives you "pieces" which can then be integrated into larger programs. You get a more efficient use of memory and a reduction in the time it takes to develop large programs.

Turbo Pascal for the Mac is so compatible with Lisa® that they should be living together

Routines from Macintosh Programmer's Workshop Pascal and Inside Macintosh can be compiled and run with only the subtlest changes. Turbo Pascal for the Mac is also compatible with the Hierarchical File System of the Macintosh.

The 27-second Guide to Turbo Pascal for the Mac

- Compilation speed of more than 12,000 lines per minute
- "Unit" structure lets you create programs in modular form
- Multiple editing windows—up to 8 at once
- Compilation options include compiling to disk or memory, or compile and run
- No need to switch between programs to compile or run a program
- Streamlined development and debugging
- Compatibility with Macintosh Programmer's Workshop Pascal (with minimal changes)
- Compatibility with Hierarchical File System of your Mac
- Ability to define default volume and folder names used in compiler directives
- Search and change features in the editor speed up and simplify alteration of routines
- Ability to use all available Macintosh memory without limit
- "Units" included to call all the routines provided by Macintosh Toolbox

3 MacWinners from Borland!
First there was SideKick for the Mac, then Reflex for the Mac, and now Turbo Pascal for the Mac™!

Suggested Retail Price: $99.95* (not copy protected)

*Introductory price expires July 1, 1987

Minimum system configuration: Macintosh 512K or Macintosh Plus with one disk drive.

BORLAND INTERNATIONAL

Borland Software ORDER TODAY

BORLAND
INTERNATIONAL

4585 Scotts Valley Drive   Scotts Valley, California   95066

To Order By Credit Card, Call (800) 255-8008

In California call (800) 742-1133

In Canada call (800) 237-1136

BOR 0234

TURBO PROLOG™ TOOLBOX

Enhances Turbo Prolog with more than 80 tools and over 8,000 lines of source code

Turbo Prolog, the natural language of Artificial Intelligence, is the most popular AI package in the world with more than 100,000 users. Our new Turbo Prolog Toolbox extends its possibilities.

The Turbo Prolog Toolbox enhances Turbo Prolog—our 5th-generation computer programming language that brings supercomputer power to your IBM PC and compatibles—with its more than 80 tools and over 8,000 lines of source code that can be incorporated into your programs, quite easily.

Turbo Prolog Toolbox features include:

- ✓ Business graphics generation: boxes, circles, ellipses, bar charts, pie charts, scaled graphics
- ✓ Complete communications package: supports XModem protocol
- ✓ File transfers from Reflex,® dBASE III,® Lotus 1-2-3,® Symphony®
- ✓ A unique parser generator: construct your own compiler or query language
- ✓ Sophisticated user-interface design tools
- ✓ 40 example programs
- ✓ Easy-to-use screen editor: design your screen layout and I/O
- ✓ Calculated fields definition
- ✓ Over 8,000 lines of source code you can incorporate into your own programs

Minimum system requirements: IBM PC, XT, AT or true compatibles. PC-DOS (MS-DOS) 2.0 or later. Requires Turbo Prolog 1.10 or higher. Dual-floppy disk drive or hard disk. 512K.

IBM® VERSION

PC, XT,® AT,® & True Compatibles

BOR 0239

BORLAND INTERNATIONAL

*4585 SCOTTS VALLEY DRIVE
SCOTTS VALLEY, CA 95066*

ISBN 0-87524-158-1