Maniplus Guide

Coordination and design	Maarten Schuerhoff Lon Hofman
Documentation	Mark Pierzchala (Westat, Inc.) Jelke Bethlehem Edwin van der Hoeven
Developers environment	Guus Razoux Schultz Erwin Kalvelagen
Parsers	Jean-Pierre Kent Leon Vermeulen Maarten Schuerhoff
Data entry program	Guus Razoux Schultz
Structure editor	Jean-Pierre Kent
Data engines	Hans Wings
Data viewer	Han Hölsgens
Manipula/Maniplus	Hans Wings Lon Hofman
Cati management	Peter Golsteijn Lon Hofman John Meertens
Meta engine & Cameleon	Leon Vermeulen
Abacus	Anco Hundepool Alex van Buitenen Aad van de Wetering
Support	Peter Stegehuis Sandra Vogelesang
Cover design	Saskia van der Linden (TRIPdesign)

Maniplus Guide
Statistics Netherlands
Department of Statistical Informatics
Voorburg

ISBN 903 572 832 7

1. Introduction

Maniplus is a powerful new addition to the Blaise family of integrated survey software used for survey management. Major uses of Maniplus include interactive survey control systems such as a CAPI laptop management system (single-, or multi-survey), an in-office data flow control system, a data editing organizing system, and top-down data review with access to micro data. Maniplus can also be used to enhance CATI management through implementation of complex survey management protocols.

Maniplus control systems can be created easily. The developer can draw on many visual elements and language features that are specifically suited for survey management. An organisation can build survey management systems in many different ways from top-down menus and several types of dialogs. Maniplus knows the meta data of Blaise instruments and can load them into memory before they are needed for a task. It can read from, display elements of, and write directly to Blaise data files. This direct access to the other parts of the Blaise system give it a performance and functionality advantage over other systems that might be used to manage Blaise surveys.

Other capabilities of Maniplus include a help system with hyper-text links and the ability to run Blaise instruments, Manipula or Maniplus setups, and an external program such as file compression software. A particularly striking feature of Maniplus is its ability to display an interactively sorted lookup table. This is especially valuable for CAPI laptop management. If an interviewer has a caseload on her laptop, she needs to be able to see the data in several ways. A lookup table can display selected fields from a Blaise data set, in this case survey management fields. She may need to sort the data in this lookup table by form status, town, post code, ID number, or other ways. Maniplus gives the ability to perform this sort on demand, allow the interviewer to scroll through the table (or to use a pointing device), and invoke the Data Entry Program on a certain record. When she exits the interview, the form status is changed if appropriate and the interactive lookup table reflects the changed status, sorting the table again if appropriate.

This manual is organized into two main parts. The first part explains the basic elements. Chapter 2 covers the menu system, Chapter 3 enumerates the many kinds of dialogs,

Chapter 4 deals with invoking other programs from Maniplus, Chapter 5 treats file handling, and Chapter 6 describes the help system.

The second main part of the manual includes example applications. Chapter 7 demonstrates a simple CAPI laptop management system, Chapter 8 suggests a way that top-down data editing could be implemented, and Chapter 9 demonstrates ways to enhance CATI management.

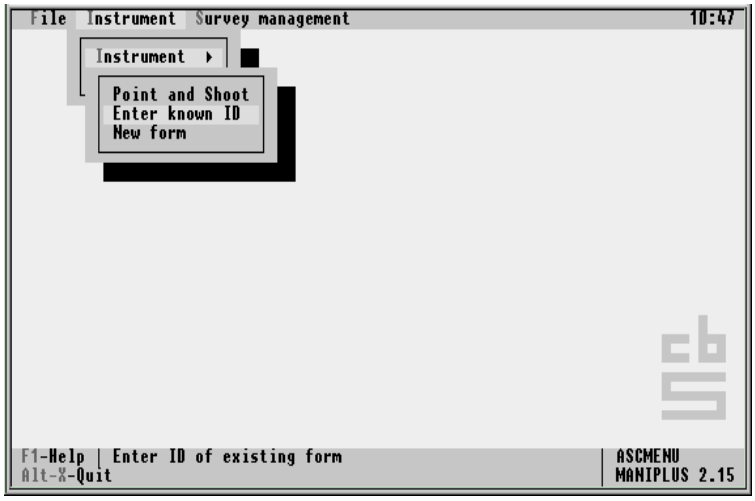
Statistics Netherlands wishes to thank Westat, Inc. for allowing Mark Pierzchala to contribute large parts of this manual and to test the software.

Statistics Netherlands also wishes to thank the Australian Bureau of Statistics for information about their CAPI laptop and in-office management system, which provided useful information for the present documentation.

2. Menus

Many Maniplus applications are implemented through menus. The menus in Maniplus are pull down menus originating from a top menu bar.

Figure 2.1. A pull down menu



Menus are an elegant and easy way for users to navigate through the application and execute tasks. They can see the relationship of each task to related tasks. You can attach hot keys and function keys to each menu selection. The status bar at the bottom of the screen can carry a longer explanation of each menu choice. A hyper-text-link help facility is easily implemented and can be used to give detailed explanations of each menu choice.

If you are in the specification stage of building a Maniplus application, it is easy to draft menu proposals, show them on screen, and allow others to comment.

To implement menus in Maniplus you need a menu file to hold menu information, a description of the menu file, one auxfield (with a second one strongly advised), and a manipulate section with commands that run the menu.

2.1 Getting Started

You can easily get started in building menus by copying any of the three example setups *ASCMENU.MAN*, *DM_MENU.MAN*, or *PROCMENU.MAN* and associated files into a working directory. The first two examples named are the easiest to start with. For example, in order to start changing the menu system for *ASCMENU.MAN*, all you need to do is to edit the contents of the file *ASCMENU.ASC*. You do not have to re-prepare the setup to see the menu choices change.

2.1.1 Using the Help Facility

The user can invoke the help facility for any menu choice, where the developer has help defined, by moving the cursor onto the menu choice and then pressing the <F1> key. The user knows where the cursor is because the menu choice is highlighted. Implementing the help facility is covered in Chapter 5.

2.2 Menu file description

The menu file description can be held in the setup itself or in a separate Blaise data model. The following example description is taken from the *USES* section of the example *ASCMENU.MAN*. You have some latitude to change the length of each field depending on the needs of the project.

```
USES
  DATAMODEL MenuDescription
    INCLUDE "PlusMenu.INC"          {Eligible hot keys}
    FIELDS
      MenuID : STRING[8]
      MenuText : STRING[20],   EMPTY
      FunctText : STRING[10],   EMPTY
      FunctKey : TFunctionKey, EMPTY
      StatusText : STRING[40],   EMPTY
      HelpID : 1..900,       EMPTY
      Program : STRING[10],   EMPTY       {Extra FIELD}
      RunParam : STRING[10],   EMPTY      {Extra FIELD}
  ENDMODEL
```

The include file *PlusMenu.INC* holds a list of eligible hot keys as a *TYPE* called *TFunctionKey*. You can just include this file in the menu description as it is provided in the Maniplus distribution but you could also make up your own. It is more efficient if the enumerated *TYPE* has only the function keys you really use in the Maniplus setup.

The description must have at least five menu elements, six if menu help is used. You can define more menu elements if the application requires it. The fields *Program* and *RunParam* are not required by Maniplus but may be by an application.

- *MenuID* is a unique number. Menu levels are separated by a period. For example:
  1, the first high level menu,
  1.2, the second submenu of the first high level menu

Maniplus requires menu ID numbers to be in ascending order though it does allow gaps between numbers. If you want the numbering to be more robust, you could adapt a numbering scheme like 1.1, 1.10, 1.20, 1.30,10.10,10.20, and so on. This leaves gaps and allows you to insert menu choices more easily.

For an ASCII menu file, if the physical order of the records in the file is 1, 1,5, 1.2, 1.3, 1.4, 1.6, then only menu choices 1, 1.5, and 1.6 would show up on the screen.

If the number of entries in a submenu is 10 or more you can have a problem if the menu ID is a primary key in a Blaise menu file. For example, 1.10 comes before 1.2. But Maniplus expects the keys in the order 1.2, 1.10. You can manage 10 or more submenu entries by using a menu ID numbering scheme using numbers like 1.02 instead of 1.2.

- *MenuText* is the text for the menu entry. Up to 20 characters are allowed for this application but you can define more characters. To designate a hot key, use '~'. For example:
  ~F~ile will highlight the F and make it a hot key,

- *FunctText* is a readable string displayed in the menu that tells the user the appropriate function key which can be pressed to invoke the menu option. For example:
  Alt-X.

- *FunctKey* is the number of the function key matching the function key description you just typed which is held in FunctText. If you use the Blaise data model *DM_MENU.BLA* to enter the menu elements, you easily enter this number from the TYPE incorporated in the data model. Otherwise, you can find the number by inspecting the *PlusMenu*.INC file. For example:
  100 is the number corresponding to Alt-X.

- *StatusText* is displayed in the status bar that gives the user more information about the menu choice the cursor is currently sitting on. For example:
  Select a form and press Enter to start interview.

- *HelpID* is a number referring to the appropriate hyper-text entry in the help file (described below and in Chapter 6). This description allows up to 900 help entries, but you can allow numbers up to 10,000.

- The field *Program* is user-defined and not required by Maniplus. In these examples, it is the name of the procedure or dialog to invoke in the Maniplus setup. For example, the case statement in the manipulate section may look (something) like:

```
CASE program OF
  'aproc' : A_Proc
  'about' : AboutDialog
ENDCASE
```

For this entry in the menu file you enter program commands like *about* or *aproc*. The case structure in the manipulate section of the setup will associate the program commands from the menu to procedures or dialogs in the setup.

You do not have to use a field such as *Program*. The case structure could be written as:

```
CASE MenuID OF
  '1.2' : AboutDialog
  '1.5' : A_Proc
ENDCASE
```

However, the use of a field like *Program* is a more robust way of implementing menu commands. The contents of the field *Program* is case sensitive (*ToBe* is not *tobe*).

- *RunParam* is a parameter if the program snippet you are invoking requires it. It is not used in this manual but is included to give the developer an idea of how this could be implemented.

If you have the menu elements below entered in a record of the menu file, corresponding to the description above,

```
MenuID     =   1.2
MenuText   =   ~A~bout
FunctText  =   Ctrl-F10
FunctKey   =   30
StatusText =   Show the About box
HelpID     =   12
Program    =   about
{no parameter}
```

then the result on the screen will look like:

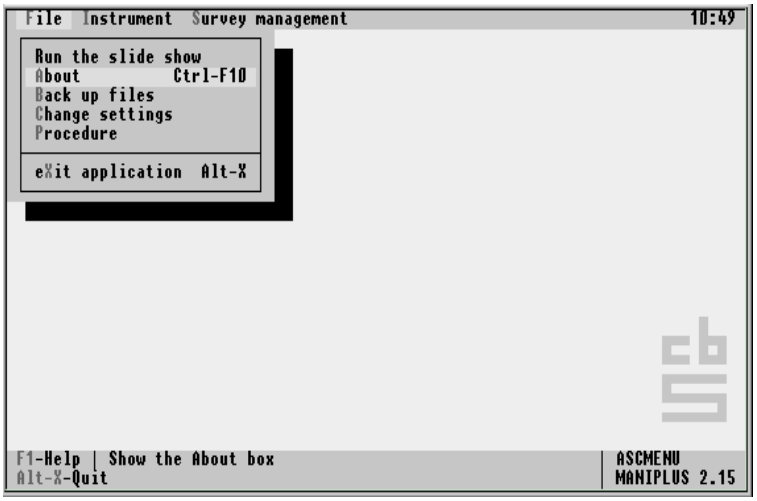


Figure 2.2.1. The About menu entry

Separation line in the menu

You can put a line in one of the pulldown submenus to group similar entries. If you only fill in the MenuId you will have a line in the submenu. In the menu file *ASCMENU.ASC*, line 1.6 is a blank line that separates the option to exit the setup from other options.

2.3 Menu File

Menu elements are held in a file that is accessed and interpreted by the Maniplus application. There is one record per menu entry in the file. The menu file can be a Blaise file, an ASCII file, or a temporary file.

There are three ways to create menu files corresponding to the kind of menu file you use. If the menu file is a Blaise data file, you invoke a Blaise data model with the Data Entry Program (DEP) to fill in menu components. If the menu file is an ASCII file, you edit it with a text editor to enter the menu elements. If the menu file is a temporary file, you declare the menu elements with write instructions from within the Maniplus setup. There are advantages and disadvantages of using each kind of file.

Advantages and Disadvantages of a Blaise Menu File

Advantages: A Blaise data model, such as *DM_MENU.BLA* guides you through the entries needed for each menu choice. You are prompted for the menu ID number and other elements. For each element, an explanation of the element is given along with an example. By pressing <Ctrl F7> (by browsing forms), you can get a nice overview of the menu elements. If you specify the ID numbers as a primary key in the Blaise data model, then duplicate ID numbers cannot be entered. When you change the menu file entries, you do not have to re-prepare the Maniplus setup for the menu changes to take effect. It is possible to implement dynamic menus with a Blaise menu file if an updatefile section is used to refer to it.

Disadvantages: Once you are familiar with the menuing scheme in Maniplus, you may find it easier to create and modify an ASCII menu file in a text editor than to go through the Blaise data model. When you use a Blaise data file to hold menu entries, you have to distribute several Blaise files with the application including instrument and data files. It can be difficult to insert

entries in a Blaise file, you'll probably have to renumber surrounding entries unless you adopted a robust numbering scheme as described above. If you do not like the menu file description provided in the data model *DM_MENU.BLA*, then you have to change and re-prepare it and then update the Blaise menu file to the new description. Using a Blaise file with an updatefile section for a dynamic menu may not perform as fast as using a temporary file (see below).

Advantages and Disadvantages of an ASCII menu file.

Advantages: The ASCII file of menu choices and elements is easy to create and maintain once you are familiar with menu requirements and possibilities. You only have to distribute one or two small files with the application instead of several larger ones as you would do if Blaise menu files were used. When you change the menu file entries, you do not have to re-prepare the setup for the menu changes to take effect. You may specify the data model in the setup instead of using a reference in the uses section to an ~MI file as you do for a Blaise menu file.

Disadvantages: The ASCII menu file is easily accessed and tampered with by anyone using a text editor. If you are using separators in the menu file, then it can be a little more difficult to maintain or create the file correctly. However, you can use an ASCII fixed format file instead of a separated file. The menu developer must make sure that the records of the ASCII file are arranged in the proper order by menu ID number (see above).

Advantages and Disadvantages of a temporary file menu file

Advantages: All menu information is held within the setup itself. There are no extra files to maintain or to distribute with the application. Among other things, this will make it easier to prevent tampering with the menu file. You do not directly enter the records in the temporary file. This is done when the Maniplus application is invoked by procedures within the Maniplus setup. You may specify the data model in the setup instead of a reference in the uses section to a ~MI file as for a Blaise file. An important advantage of temporary files is dynamic menus. If you define the menu dynamically in the setup you can easily come up with a menu depending on type of user or the language of the

user. It is even possible to change the menu during a session, to add or delete menu entries depending on previous user activity. A more experienced Maniplus developer will probably, in the end, prefer the use of the temporary menu file. See below for an example of a dynamic menu in the setup *DYNAMENU.MAN*.

Disadvantages: It can be more difficult to create and maintain the menu records within the setup due to the way the menu records are typed. In order to change the menu information, you have to re-prepare the setup. If a parameter is out of place, you may have to go through a few iterations of preparing the setup to get the menu procedures to pass the syntax check. However, this is probably more a matter of experience.

For distributed applications, for example a laptop CAPI management system, it is recommended that you use a temporary file as a menu file. The minor inconvenience of working with procedure statements in the setup are outweighed by the fact that there are fewer files to distribute and maintain, and the fact that other people cannot tamper with the menus. As applications grow, you may find the use of dynamic menus to be necessary, and if implemented in temporary menu files, to be easier to create and maintain.

Changing the type of menu file

If you start by using a Blaise data file as a menu file and would like to change to an ASCII menu file, you can use the setup *TOASMENU.MAN*. If you wish to go from an ASCII menu file to a Blaise menu file, use the setup *TOBLMENU.MAN*. A similar Manipula setup that recasts the contents of *ASCMENU.ASC* procedure statements needed for a temporary file is *TOPRMENU.MAN*. It creates the file *CreateMU.INC* which holds the procedure *CreateMenu*. The setup *PROCMEN2.MAN* includes this file in order to create the menu from a temporary file.

2.4 Declaring the menu file

Having described the menu file in the USES section, you must declare or name the menu file either with an inputfile, updatefile, or a temporaryfile section. For example from the example setup *ASCMENU.MAN*:

```
INPUTFILE MyMenu :
  MenuDescription ('AscMenu.ASC', ASCII)
```

See the example setups *PROCMENU.MAN* and *DYNAMENU.MAN* to see how the key word temporaryfile is used when a temporary file is the menu file and procedures within the setup create the menu. The difference between using a temporaryfile and an updatefile is that the former is stored in memory and not saved when the setup is stopped. The updatefile resides on disk, and any information in it that has been changed in the course of the use of the setup is saved. The use and updating of a temporaryfile is faster than an updatefile. On the other hand, there may be advantages to saving current menu lines in a Blaise file through updatefile.

2.5 AUXFIELDS Used in Maniplus Menus

In a well designed menu system, you need two auxfields in the setup. They can have any name, but *Reslt* and *Stop* used in the example setup *ASCMENU.MAN* convey their meanings.

```
AUXFIELDS
  Reslt     : INTEGER

  Stop : (yes, no)
```

These auxfields are used in the manipulate section to run the menu robustly. *Reslt* is always used, the use of *Stop* is strongly advised.

- *Reslt* holds the result of a test performed by Maniplus. If the menu file can be found, *Reslt* will equal zero, otherwise it will have a non-zero value. You test on the value of *Reslt* to ensure the menu file is in place before invoking the menu.

- *Stop* is used to give the user a second chance when he tries to leave the menu. He may have hit the exit menu choice accidentally. If so, the *Stop* confirmation will give the user a chance to stay in the menu system.

2.6 MANIPULATE Section

The menu system is run within the manipulate section. There are several aspects of running the menu that are shown in the code below. If you wish, you can just copy this code from one of the example setups and use it directly without bothering with the details. The following is taken from the example setup *ASCMENU.MAN*.

```
MANIPULATE
  REPEAT
    Reslt := MyMenu.MENU('~Alt-X~-Quit')
    IF (functkey<>altx) and (Reslt=0) THEN
      CASE program OF
      {from menu script}    {from PROCEDURE above}
         'aproc'          :     A_Proc
         'about'          :     AboutDialog
      ENDCASE
    ENDIF
    IF functkey=altx THEN
      IF NOT (CONFIRM('Are you sure you want to quit?'))
      THEN
        Stop := no
      ELSE
        Stop := yes
      ENDIF
    ENDIF
  UNTIL (functkey=altx) AND (Stop = yes)
```

To change functionality from application to application, all that you need to change in the code above is the details of the case statement. This manipulate section will check that the menu file is in place, that the <Alt-X> key has not been pressed by the user, associate a menu choice with a dialog or procedure in the setup, run the menu, allow the user to exit by pressing Alt-X, and give the user a second chance to stay in the menu system with a confirmation window. See any of the three example setups *ASCMENU.MAN*, *DM_MENU.MAN*, or *PROCMENU.MAN* for more information on this manipulate section.

2.7 Using a TEMPORARYFILE to Hold the Menu File

A temporaryfile is a file created in memory by the setup at the time the setup is running. By using temporaryfile to hold the menu file you can discourage tampering with the menu definitions and can implement dynamic menus through the temporary menu files which are held in memory and can be changed by the setup. The examples *PROCMENU.MAN* and *DYNAMENU.MAN* are examples of the use of a temporary menu file. The menu file records are written in the setup itself with WRITE instructions. In the examples the WRITE instructions are implemented in a procedure called *MakeMenuLine*. They give instructions on how to write one line of the menu.

```
PROCEDURE MakeMenuLine(ID : STRING;
                       MT : STRING;    {parameters}
                       FT : STRING;
                       FK : INTEGER;
                       ST : STRING;
                       HI : INTEGER;
                       PR : STRING;
                       RP : STRING)

  MyMenu.MenuID:= ID                   {assignments}
  MyMenu.MenuText:= MT
  MyMenu.FunctText:= FT
  MyMenu.FunctKey:= FK
  MyMenu.StatusText:= ST
  MyMenu.HelpID := HI
  MyMenu.Program:= PR
  MyMenu.RunParam := RP
  MyMenu.WRITE                      {WRITE instruction}
ENDPROCEDURE
```

The menu itself is generated in another procedure called *CreateMenu* which makes repeated calls to the procedure *MakeMenuLine*, passing appropriate parameters each time. Only part of the procedure from *PROCMENU.MAN* is shown below.

```
PROCEDURE CreateMenu
  MakeMenuLine('1','~F~ile','F-2',2,'File menu',1,'','')
  MakeMenuLine('1.1','Run the slide show','',0,
    'Slide Show explanation of the survey',0,'show','')
  MakeMenuLine('1.2','~A~bout','Clt-F10',30,
    'Show the About box',1,'about','')

 . . .

ENDPROCEDURE
```

Each menu element is a parameter that is passed to the other procedure called *MakeMenuLine*. The menu elements (parameters) are separated in this example by commas. An element which is a string is started and finished by a straight single quote (').

The procedure *MakeMenuLine* is the same in both *PROCMENU.MAN* and *DYNAMENU.MAN*. The procedure *CreateMenu* changes from setup to setup.

In the manipulate section, the procedure *CreateMenu* is the first command invoked.

```
MANIPULATE
  CreateMenu
```

2.8 Dynamic Menus

It is possible to implement dynamic menus in Maniplus. There can be one menu for supervisors and another for interviewers, or there can be menus for English, Dutch, French, German, Spanish, or any language. You can make menu choices appear or disappear, or enable and disable them, depending on the user actions. All of these possibilities are demonstrated with the example setup *DYNAMENU.MAN*.

2.8.1 Menu File

The dynamic menu file can be a temporary file held in memory.

```
TEMPORARYFILE MyMenu : MenuDescription
```

This allows the fast updating of the records in the menu file including adding and deleting them, or destroying the file altogether and instantly recreating it.

It is also possible to use a Blaise file through an updatefile section as a dynamic menu file, but it may not perform as fast as the temporary menu file. All examples of dynamic files in this manual are with temporary menu files.

2.8.2 Alternate Menus

An application may require different menus for different kinds of users. For example, a supervisor and an interviewer may have different kinds of tasks to carry out, your study may employ

interviewers who speak different languages, or one menu may be a subset of another. If the two different kinds of users share a lot of tasks (procedures) then it will be very efficient from a programming point of view to make one setup act both ways. To have alternative menus from one setup it is necessary to create them dynamically from the setup. This means that the temporaryfile is created by a procedure. For two alternate menus, two procedures, are programmed, one each for making one of the versions of the menu file. In *DYNAMENU.MAN* two procedures that do this are called *InterMenu* and *SuperMenu*. Part of *InterMenu* is shown below:

```
PROCEDURE InterMenu
  MakeMenuLine('1','~F~ile','F-2',2,'File menu',1,'','')
  MakeMenuLine('1.1','Run the slide show','',0,
     'Slide Show explanation of the survey',0,'show','')
 . . .
ENDPROCEDURE
```

Both procedures make use of the procedure *MakeMenuLine* displayed above in the *PROCMENU.MAN* example.

Alternating between menus

In the manipulate *U.MAN* this is done with an IF statement involving the auxfield *UserType*.

```
UserType := DOSENV ('UserType')

IF UserType = 'super' THEN
  SuperMenu
ELSEIF UserType = 'inter' THEN
  InterMenu
ELSE
  DISPLAY('The DOS environment variable UserType is not
         set properly.', WAIT)
ENDIF
```

UserType gets its value from a DOS environment variable of the same name with the *DOSENV* function. If *UserType* is set to 'super' then the supervisor's menu is invoked. If it is set to 'inter' then it is set to the interviewer's menu. If the DOS variable is not

set then the setup displays a message and quits without displaying a menu.

The DOS environment variable can be set in a BAT file before the setup is invoked, or it can be set in the *AUTOEXEC.BAT* file if supervisors and interviewers have different ones installed on their machines. If it is set in the *AUTOEXEC.BAT* file, then the DOS environment space will expand automatically to accommodate it if necessary. If it is set in another BAT file, then you must ensure that there is enough environment space to accommodate the new DOS environment variable.

Two BAT files have been programmed for this example. They are called *SUPER.BAT* and *INTER.BAT* respectively. The latter is displayed.

```
set usertype=inter
call maniplus dynamenu.man
set usertype=
```

This BAT file will invoke the setup with the interviewer menu, the other will invoke the setup with the supervisor's menu.

The auxfield *UserType* can get its value by other methods. For example, instead of using a DOS environment variable, the setup could get it from an interviewer profile file.

Dynamen2.man

A second method of designating the kind of user is demonstrated in the setup *DYNAMEN2.MAN*. There, a dialog is invoked where the user can make a choice of user type.

2.8.3 Modified Menus

Instead of choosing between two or more menu files, you can dynamically modify one menu file. It is possible to add or delete

menu choices or to enable and disable them. Both kinds of menu modifications are demonstrated in *DYNAMENU.MAN*.

Adding and Deleting menu choices

To add a menu choice, a record must be added to the menu file. To delete a choice, a record must be deleted from the menu file. Adding a record can be done with the procedure *MakeMenuLine* (see above). Deleting a record can be done with the procedure *DeleteMenuLine*.

```
PROCEDURE DeleteMenuLine (ID : STRING)
  MyMenu.GET(ID)
  MyMenu.DELETE
ENDPROCEDURE
```

It now remains to call either *MakeMenuLine* or *DeleteMenuLine* under the appropriate circumstances. The procedure A_Proc1 alternately adds and deletes the menu choice for the About box.

```
PROCEDURE A_Proc1
  MyMenu.GET('1.2')
  IF MyMenu.ResultOk THEN
    DISPLAY('The menu for the About box will be
             deleted.', WAIT)
    DeleteMenuLine('1.2')
  ELSE
    DISPLAY('The menu for the About box will be added.'
           , WAIT)
    MakeMenuLine('1.2','~A~bout','Clt-F10',30,
                 'Show the About box',1,'about','')
  ENDIF
ENDPROCEDURE
```

The *GET* method in conjunction with the *ResultOK* function checks if there is a record in the menu file with *MenuID* '1.2'. If so, then the procedure *DeleteMenuLine* deletes it. If the line is not in the menu file, then the procedure *MakeMenuLine* is invoked to create it with the necessary information passed as parameters. The menu is instantly updated when Maniplus leaves the procedure *A_Proc1*.

Figure 2.8.1.
With the About entry

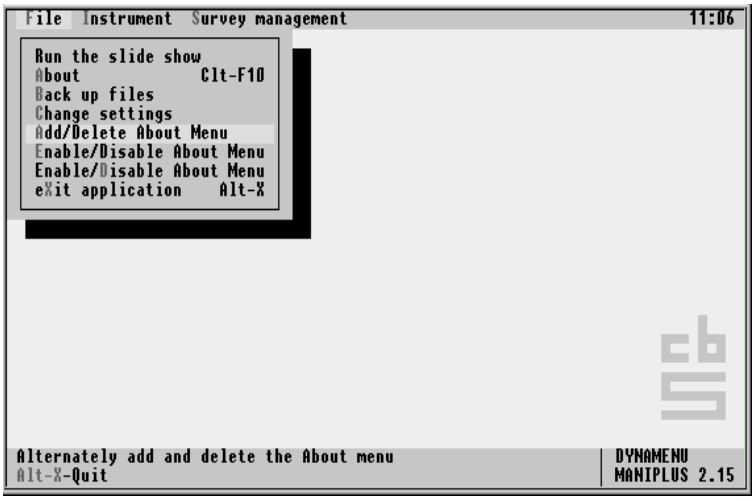
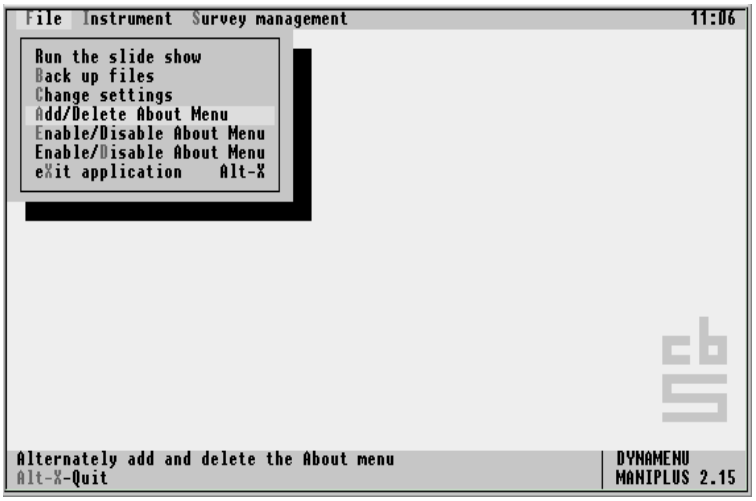


Figure 2.8.2.
Without the About entry



You can execute the setup *DYNAMENU.MAN* to see this work interactively.

Enabling and Disabling menu choices

To either enable or disable a menu choice it is necessary to modify the corresponding line in the menu file. This can be done, for both enabling and disabling a menu choice, using the procedure *MakeMenuLine*. The procedure, *A_Proc2* alternately enables and disables the menu

choice for the About box using the *MakeMenuLine* procedure. To disable a menu line in this example, it is sufficient to change or delete the field *Program* in the menu line for the specific record. For example, for *MenuID* 1.2, the menu line can be disabled by changing the field *Program* from 'about' to anything else. This is because the case structure in the manipulate section will look specifically for 'about'.

If a menu choice is disabled, then it is good practice to let the user know this when looking at the menus. It is not possible to 'grey out' menu choices in Maniplus as you can in other menuing systems. However, it is possible to adopt a similar convention and accomplish the same thing. The convention adopted for this example is to place a minus sign as the first character in the *MenuText* and to disable the hot key. The function key assignment is also removed. This is all done with *MakeMenuLine*.

```
PROCEDURE A_Proc2
  MyMenu.GET('1.2')
  IF MyMenu.ResultOk THEN
    IF MyMenu.Program = 'about' THEN
      MakeMenuLine('1.2','- About','',0,
                   'Show the About box',1,'','')
    ELSE
      MakeMenuLine('1.2','~A~bout','Clt-F10',30,
                   'Show the About box',1,'about','')
    ENDIF
  ELSE
    MakeMenuLine('1.2','~A~bout','Clt-F10',30,
                 'Show the About box',1,'about','')
  ENDIF
ENDPROCEDURE
```

If the line in the menu file with *MenuID* 1.2 is present, then the procedure checks to see if the line (i.e., menu choice) is presently enabled or disabled. It does this by checking the value of the field *Program*. If *Program* is equal to 'about' then the line is considered enabled, otherwise it is considered disabled. If the menu choice is enabled, then it is disabled by setting *Program* = '', and the user is informed of this by setting *MenuText* = '-

About' and *FunctText* = ''. This is done with the first *MakeMenuLine* call above.

If the menu choice is currently disabled, then it is enabled through the second call to *MakeMenuLine* shown in this example. The same call to *MakeMenuLine* is used to create the menu line if it does not exist.

It is possible to enable and disable menu choices without the procedure *MakeMenuLine*. This is done by making direct assignments to fields in the menu file. This is demonstrated in the procedure *A_Proc3*, only part of which is shown.

```
MyMenu.GET('1.2')
  IF MyMenu.ResultOk THEN
    IF MyMenu.Program = 'about' THEN
      MyMenu.MenuText := '- About'
      MyMenu.FunctText := ''
      MyMenu.FunctKey := 0
      MyMenu.Program := ''
      MyMenu.WRITE
  . . .
```

A_Proc2 and *A_Proc3* have the same functionality but accomplish the task differently.

2.9 Specifying a CAPI Laptop Menu System

A complete laptop interviewing menu system is held in the Maniplus setup *CAPILAP1.MAN*. None of the menus, other than the *About* dialog, do anything except display a message of what they will do when the system is implemented. This setup will be revised later into a complete laptop management system.

2.10 Two or more languages

Maniplus does not have a natural way to implement two or more languages. However, you can implement three dual-language features, two of them very easily.

- To implement menus in different languages, use dynamic menus described above. From a DOS environment variable, a dialog, or in a profile file, state or determine the language of the user. Then invoke the correct file from the user setting.
- To implement menu help in two languages in the same help file, keep two sets of codes for the same menu entry. For example, if you have an English entry with menu code 101, you can set a corresponding French entry with code 8101. Then when you switch menus as described just above, you'll also switch help file codes and the user will have help in his own language. See Chapter 5 for details on the help facility.
- To have dual-language buttons, it is necessary to define two auxfields, one for the first language, one for the second. The Maniplus code for procedures and dialogs will be in terms of the first auxfield. When a button from the second language is pressed, define a computation that gives the corresponding value to the first button-related auxfield, which will then invoke the proper procedure or dialog. See Chapter 3 on Dialogs for information on buttons.

To take the idea of dual languages further in Maniplus, you would have to doubly define dialogs in two languages. The various language dialogs could still refer to the same procedures.

2.11 Example Files About Menus

The files are listed by extension and by general purpose.

File Name	Description
ASCMENU.ASC	ASCII menu file.
INTER.BAT	BAT file that sets the DOS variable UserType to 'inter' and then invokes *DYNAMENU.MAN*
SUPER.BAT	BAT file that sets the DOS variable UserType to 'super' and then invokes *DYNAMENU.MAN*
DM_MENU.BLA	Blaise data model that describes the menu file. It can be used to enter menu file elements.
CREATEMU.INC	An included file created by the setup *TOPRMENU.MAN*. This is used by the setup *PROCMEN2.MAN*.
ASCMENU.MAN	A setup that uses an ASCII file as a menu file.
DM_MENU.MAN	A setup that uses a Blaise file as a menu file. DM stands for data model.
PROCMENU.MAN	A setup that uses a temporary file as a menu file.
DYNAMENU.MAN	A setup that illustrates how to make dynamic menus. It uses temporary files.
DYNAMEN2.MAN	The same as *DYNAMENU.MAN* but uses a dialog to determine the user's status.

PROCMEN2.MAN	A setup that uses a temporary file as a menu file and includes the file CreateMU.INC.
CAPILAP1.MAN	A setup that specifies the menu system of a complete laptop interviewing menu system.
TOASMENU.MAN	A setup that converts a Blaise menu file to an ASCII menu file.
TOBLMENU.MAN	A setup that converts an ASCII menu file to a Blaise menu file.
TOPRMENU.MAN	A setup that converts an ASCII menu file to an included file CreateMU.INC that is used to create a temporary menu file in the setup *PROCMEN2.MAN*.
ASCMENU.TXT	An ASCII file of help text that is converted to *ASCMENU.HLP* and is used by some setups in this section for a help facility.

3. Dialogs

Dialogs are one of the main ways that the user interacts with a Maniplus application. With dialogs you can display text information, display a file or a report and let the user browse through it, obtain information from the user, enter data directly into a Blaise data set, invoke a procedure or another dialog, select a form in the Blaise data set to process, and create interactive tables that allow users easy access to groups of forms. All of these possibilities and more are demonstrated in the example setups *DIALOGS.MAN*, *DIALPROC.MAN*, and *LOOKTEMP.MAN*. The programmer comments in each informs you how to prepare and run them.

DIALOGS.MAN contains 22 dialogs showing all possibilities in developing, arranging, and controlling their behaviour. Except for the demonstrations of lookup dialogs, these example dialogs do not interact with data files. There are no procedures in this example.

DIALPROC.MAN contains 16 dialogs and procedures and shows how they work together. This example, in addition to showing more about constructing dialogs, demonstrates how to call up the data entry program (DEP) from Maniplus, how to display data from a Blaise data set in the dialog, and how to write information to the Blaise data set from Maniplus.

LOOKTEMP.MAN shows how to make very powerful lookup dialogs using temporary files. This combination allow you to sort lookups as well as to display data from a subset of forms. It also demonstrates some performance enhancing techniques with secondary keys and filters. There are 14 dialogs and procedures in this setup.

A master Maniplus setup, *DIALDEMO.MAN*, organises these three setups into one menu system. You can view all of the functionality of the three examples setups from this *DIALDEMO.MAN*. It calls the other setups at your discretion. You can also invoke each of the setups individually. To run *DIALDEMO.MAN*, prepare the following (do not invoke them). For the Maniplus and Manipula setups, use the <F7> key in the control centre. For the Blaise data models use the <F9> key in the control centre.

- Prepare *NCS_EASY.BLA*.
- Prepare *DM_MENU.BLA*.
- Prepare *ASCTOSEP.MAN*.
- Prepare *DIALOGS.MAN*.
- Prepare *NCS_DATA.MAN*.
- Prepare *DIALPROC.MAN*.
- Prepare *LOOKTEMP.MAN*.
- Prepare and invoke *DIALDEMO.MAN*.

Study the examples

It is strongly advised that you make a study of the examples as you read this chapter. This means preparing, invoking, and experimenting with them. It is in this way that you get the most out of the program.

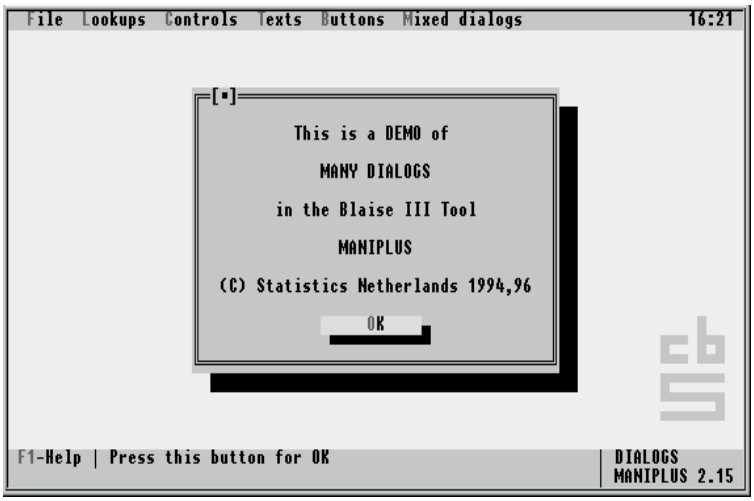
3.1 Examples of Four Dialog Elements

There are four kinds of dialog elements in Maniplus. They are *text*, *control*, *button*, and *lookup*. In the example *DIALOGS.MAN* there is a major menu heading for each of these dialog elements. You can invoke the main menu headings and their sub-menus to see how each of these elements can be used.

Example of Text

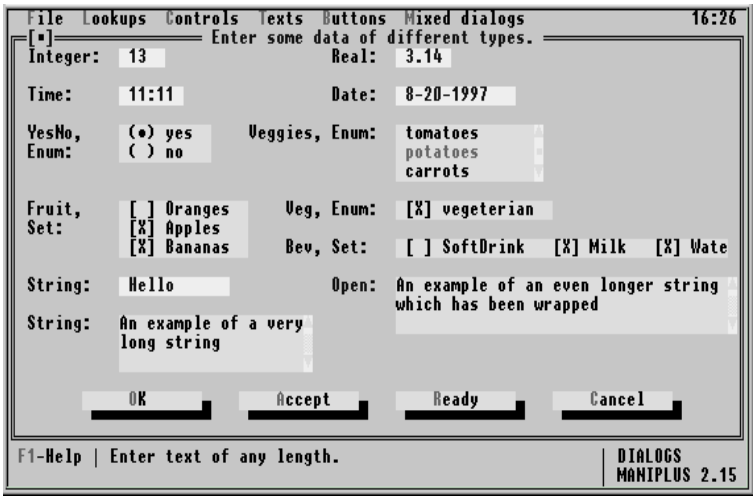
With a text dialog element you can display any fixed text information. For example, it is always a good idea to have an *About* box in a Maniplus setup.

Figure 3.1.1. The About box



With a control a user can enter information needed by an application. This may be information needed by Maniplus, for survey management, or to type data into a Blaise data set. A control can also be used to display information that the user cannot change, and it can be used to display variable text information. Any kind of data that can be entered in a Blaise Data Entry application can be entered and displayed in a control.

Figure 3.1.2. ‘ Enter Some Data’ dialog



Kinds of fields shown here include integer and real numbers, enumerated and set fields, text strings, open answers, and time and date fields.

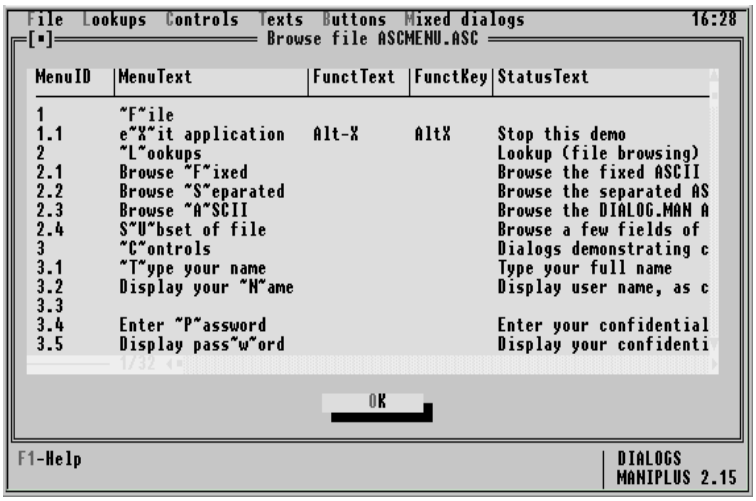
Example of Button

A button can be used to accept data entered into a control, or to reject it. It can be used to invoke another dialog or procedure. Buttons rarely appear in a dialog by themselves. The dialog displayed above in the control example has four buttons, *OK, Accept, Ready,* and *Cancel,* which do different things.

Example of Lookup

A lookup invokes a data viewer on a file. The data viewer displays the contents of a file or a Blaise data base, each field taking one column. It can display all fields or a subset of them. If the display is too large for the screen, the user can scroll right and left, and up and down. It is possible to sort the contents of the lookup and to select a form to process by scrolling to a line of the lookup and pressing <Enter> or a button. The data viewer used in lookup is capable of displaying a separated ASCII text file in a columnar display.

Figure 3.1.3.
‘ Browse fixed’ dialog



Mixing Dialog Elements

It is possible to use any combination of dialog elements in a dialog. In *DIALOGS.MAN* there is a major menu heading *Mixed Dialogs*. The sub-menu choice *Details Screen* reveals a dialog that contains all four elements.

Size and Placement of Dialog Elements

You can control the size and placement of all dialog elements and their propertiess. If you do not do this then Maniplus will attempt to do it for you. When placing two or more elements on one screen, it is advantageous to take control of the arrangement of the dialog elements. See below for details.

Behaviour of elements

You can control the behaviour of some of the dialog elements, for example, whether a user can leave the dialog with the <Escape> key, or whether an entry in a control is validated immediately after entry or when a button is pressed. See below for details.

3.2 System Provided Dialogs

Maniplus provides two dialogs that handle complicated tasks. A file listing dialog is generated when the key word *SELECTFILE* is used. This allows the user to select a file (possibly from files with certain extensions) to browse or process. In the example *DIALOGS.MAN* the sub-menu choice *Browse Any File* under *Mixed Dialogs* makes use of *SELECTFILE*.

The button properties *ONPRESS = SEARCH* generates a dialog that searches for a form within a file. This feature is demonstrated in the example *DIALPROC.MAN*. This is the *Button stuff with browse (Lookup)* sub-menu choice under the *Browse Files* main menu choice. Once the lookup is displayed, click on the search button or press <Alt-S> to invoke this form searching dialog.

Two other commands also generate windows that can be considered dialogs. These are *CONFIRM* and *DISPLAY*. In *DIALOGS.MAN*, the sub-menu choice *Display Password* under *Controls* shows a window generated by *DISPLAY*. The confirmation the user sees *(Are you sure you want to quit?)* when exiting any of the example setups is generated with the *CONFIRM* key word.

3.3 Dialog Section

A dialog section defines one dialog. There can be many dialog sections in a Maniplus setup. Each dialog section starts with the keyword dialog and ends when another dialog section, procedure, prologue, or manipulate section is encountered. It is possible to mix dialog sections with procedures. All dialog sections are placed after the file and auxfield sections and before the prologue and manipulate sections of the setup. The Blaise III Reference Manual and the on-line help in the Control Centre both give a full explanation of the syntax and other technical details of a dialog section. The phrases 'dialog section' and 'dialog' are often used interchangeably.

Heading and Settings

The dialog *EnterData* in the example setup *DIALOGS.MAN*, demonstrates the start of a dialog section with a heading and all possible properties expressly stated.

```
DIALOG EnterData "Enter some data of different types."
  POSITION = (0, 0)
  SIZE = (80, 22)
  HELP = 38
  DEFAULTBUTTON = NO
  VALIDATEDIRECT = YES
  ESCAPE = YES
  {Dialog elements follow.}
```

The name *EnterData* is the *DialogID* . This is the identifier called in the manipulate section whenever this dialog is invoked. The *DialogID* is required.

The *DialogTitle* is the string "*Enter some data of different types*". This header appears at the top of the dialog when it is displayed on the screen. The *DialogTitle* is optional.

The properties *POSITION* and *SIZE* together control the display of the dialog on the desktop. The desktop is the whole computer

screen except for the top menu bar and the two bottom status bars. *POSITION = (0, 0)* places the upper left corner of the dialog *EnterData* in the upper left corner of the desktop. The coordinates of the *POSITION* setting are the number of columns and rows from the upper left corner of the desktop. *POSITION = (3, 2)* would place the dialog's upper left corner in the 3rd column from the left and 2nd row from the top of the desktop. *SIZE = (80, 22)* gives the dialog 80 columns and 22 lines. This large dialog takes up the whole desktop. The size and position properties could have been left out as they are the default.

The dialog has entry 38 in the help file. The topic associated with number 38 is displayed whenever the user is in the dialog and presses <F1>. See the Chapter on Menus for more details about the help facility.

The properties *DEFAULTBUTTON = NO* turns off a Maniplus convention concerning the <Enter> key. Normally, the <Enter> key is interpreted by Maniplus to activate the first button in the dialog. Where there are several data items to be filled in by the user, the default use of the <Enter> key can be awkward. Users might press <Enter> thinking they are going from one data entry cell to the next only to find themselves exiting the dialog too soon. When *DEFAULTBUTTON = NO* the <Enter> key does not activate the first button. It moves the user to the next accessible dialog element whatever that may be. The default properties is *DEFAULTBUTTON = YES*, which is appropriate for many smaller dialogs. The <Tab> key should be thought of as the normal way to navigate in a dialog. This is a Windows convention that Maniplus emulates.

The properties *VALIDATEDIRECT = YES* ensures that each data item is validated as the user exits the control. Normally Maniplus checks the appropriateness of the entered data only when all controls are filled in and a button is pressed. In other words, the default property is *VALIDATEDIRECT = NO*. By using *VALIDATEDIRECT = YES*, the validation of each data item is carried out immediately. For example, if a data item range is

between 1 and 100 and the user enters 200, *VALIDATEDIRECT = YES* will notify the user of the error immediately, not waiting until the user tries to leave the dialog.

The properties *ESCAPE = YES* need not have been written here because it is the default. This properties allows the user to leave the dialog with the <Escape> key. If the developer sets *ESCAPE = NO*, then a way of leaving the dialog must be provided by the developer. This can be done with a button. If you make a dialog with no buttons and *ESCAPE = NO* then you can never leave the dialog!

When the user is allowed to leave a dialog with the <Escape> key, none of the data entered into the dialog are saved. Disabling the <Escape> key in a dialog can be a way of forcing the user to enter some data. This option should be used with care. A dialog in the example *DIALOGS.MAN*, where the properties *ESCAPE = NO* is used, is *SeparatedData* invoked from sub-menu *Browse Separated* under the main menu choice *Lookups*. In that dialog you have to press the *OK* button to leave the dialog.

3.4 Text Dialog Element

A text dialog provides the possibility of displaying text at a designated position in the dialog. This text can be instructions to the user or clarifying text. Text can be displayed anywhere on the screen within the dialog. It can be placed next to other elements such as lookup, control, or button elements. The text element is simply defined.

```
TEXT = (S, X, Y)
```

S is a text string. It is fixed, meaning you cannot substitute one text for another. *X* and *Y* are position parameters defining the number of columns and rows the text should be displaced from the upper left corner of the dialog. You can do some basic formatting with two key combinations. @> will centre the text in the dialog, starting from the value of X (which is usually *X = 1* in this case). *@/* indicates that any following text is continued on the next line.

The sole use of the dialog *AboutDialog* (displayed above) is to declare the purpose of the Maniplus setup *DIALOGS.MAN*.

```
DIALOG AboutDialog
    HELP = 41
    SIZE = (40, 15)
    TEXT = ('@>This is a DEMO of ',1,2)
    TEXT = ('@>MANY DIALOGS',1,4)
    TEXT = ('@>in the Blaise III Tool',1,6)
    TEXT = ('@>MANIPLUS',1,8)
    TEXT = ('@>(C) Statistics Netherlands 1994,96',1,10)
    BUTTON OneOKButton
      VALUE = ok
```

This dialog is 40 columns wide and 15 rows deep. It is centred on the desktop because the setting *POSITION* is not used. All the text lines of the dialog are centred because of the key

combination @> which starts them. The lines are placed on rows 2, 4, 6, 8, and 10 of the dialog. The only things the user can do with this dialog is to read it and leave it by pressing the *OK* button or <Escape>. The dialog *AboutDialog* invoked by the sub-menu choice *About* under main menu choice *Texts*.

Control Labels

Another way to place fixed text on the dialog screen is to use a control label. This is covered below.

Placing Variable Text in the Dialog

To place variable text on the screen use a control with properties *EDIT = NO*, *DISPLAY = YES*, and possibly a label to display preceding fixed text. The control can be a string or enumerated auxfield or field (from a Blaise data set). The dialog *ShowSomeButtons* from the setup *DIALOGS.MAN* uses a control to display variable text. Its auxfield is an enumerated type and it is the auxfield's label that is displayed. From the same setup, the dialog *DisplayData* displays variable text that is entered in the dialog *EnterData*.

3.5 Control Dialog Element

A control is used to enter or display the contents of a field or an auxfield. With a control, it is possible to modify selected data fields in a Blaise data set without invoking the Data Entry Program (i.e., the survey instrument). You can also use a control to obtain information from a user. An example of this latter use of a control is from the dialog *AskPassword* in *DIALOGS.MAN*. This dialog shows the use of several properties of control.

```
CONTROL Password
  LABEL      = ('User password:  ', 2, 3)
  POSITION   = (20, 3)
  EDIT       = YES
  DISPLAY    = NO
  STATUS     = 'Please enter your password'
  HELP       = 601
```

Password is the *FieldID* of the control. In this case it is an auxfield though it could also have been a field from a data model. The definition of *Password* in the auxfields section defines the length of the control in this dialog, here a *STRING* of 8 characters. It is possible to explicitly set a *SIZE*, shorter or longer than the length of the field or auxfield. If you set it longer, the user may enter an invalid answer, or it may be truncated.

The label and the *POSITION* are displayed together on line 3. The label starts in the 2nd column of the dialog, the control in the 20th. The label gives the user readable text associated with the control.

The properties *EDIT = YES* allows the user to type something into the control, in this case, a password. *DISPLAY = NO* means that the password will not be displayed as it is typed, instead, asterisks will appear, one for each typed character.

Figure 3.5.1.
'Enter Password' dialog

The properties *STATUS = 'Please enter your password'* places this text in the status bar at the bottom of the screen. If the user needs further information, the *HELP = 601* refers to an entry in the help file that can be accessed with the <F1> key.

Dialog *EnterData*

The dialog *EnterData* has 12 controls for various data types and demonstrates how they can be arranged and displayed depending on the settings given the propertiess. This dialog is in the example setup *DIALOGS.MAN*. The data definitions for the auxfields used in the controls are as follows.

```
AUXFIELDS

  AnInteger : 1..100, EMPTY
  AReal : 0..99.9, EMPTY
  TimeQuestion : TIMETYPE
  DateQuestion : DATETYPE
  YesNo : (yes, no)
  Veggies : (tomatoes, potatoes, carrots)
  Fruit : SET OF (Oranges, Apples, Bananas)
  Veg : (vegetarian)
  Beverage : SET [2] OF (SoftDrink, Milk, Water)
  AString : STRING[10]
  ALargeString: STRING
  AnOpen : OPEN
```

The controls in this example all represent auxfields but they could have represented fields from a data file. In this example, controls for *AnInteger*, *AReal*, *TimeQuestion*, and *DateQuestion* take their SIZE from their data definition of the auxfields section.

Controls for Enumerated Types

Controls for the enumerated types *YesNo*, *Veggies*, and *Veg* can be displayed as radio buttons (the default), lists of labels, or as check boxes. The properties *APPEARANCE* controls the manner of display. Radio button and label list displays require an entry, even if the underlying field or auxfield can accept empty. With a check box display, the user can leave the control empty if the underlying field or auxfield allow it, otherwise the control will have to be filled in. In a check box, the user chooses a response by placing the cursor on it and pressing <Space Bar>.

```
CONTROL YesNo
  LABEL      = ('Enum:', 1, 6)
  POSITION   = (12, 5)
  EDIT       = Yes
  STATUS     = 'Enter a Yes or No response.'

CONTROL Veggies
  LABEL      = ('Veggies, Enum:', 25, 5)
  POSITION   = (42, 5)
  EDIT       = YES
  STATUS     = 'Select a vegetable.'
  APPEARANCE = LIST
```

Controls *YesNo* and *Veggies* are enumerated types. *YesNo* appears on the dialog screen as a cluster of radio buttons because that is the default appearance. *Veggies* appears on the screen as a list of labels. For either one, once you are in the control you use the arrow key to make a choice. Leaving the control chooses the enumerated label where the cursor was sitting. The user has to choose an answer for enumerated fields or auxfields displayed as radio buttons or a list of labels, even if they are defined with the *EMPTY* value. If you want to let the user leave an enumerated field or auxfield empty, then use the checkbox display for enumerated types.

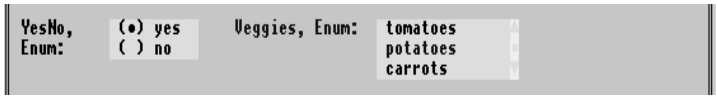


Figure 3.5.2. Enumerated fields in a dialog

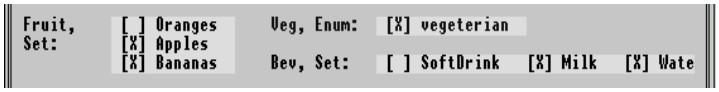
Set types and check boxes

The controls *Fruit*, *Veg*, and *Bev* are displayed in the dialog as check boxes. *Fruit* and *Bev* are displayed this way because they are set types. *Veg* is displayed this way because it is an enumerated type of only one choice. You can also display larger enumerated types as check boxes. For check box controls, the user arrows to a response label and presses <Space Bar>, or clicks on it with a mouse. *Bev* is a set auxfield of up to 2 choices. The user can choose up to 2 of its response labels.

```
CONTROL Bev
  LABEL      = ('Bev, Set: ', 29, 11)
  POSITION   = (42, 11)
  SIZE       = (36, 1)
  EDIT       = YES
  STATUS     = 'Select no more than 2 beverages.'
```

The *Bev* control makes use of the size properties. Since it has height 1, the control is arranged horizontally instead of vertically as *Fruit* is. (Other controls that make use of the SIZE properties to influence display are *ALongString* and *AnOpen*.)

Figure 3.5.3. Checkboxes in a dialog



Because the dialog *EnterData* has many elements in it, every element has an explicit position to make a presentable screen.

Order of Controls

When proceeding from one control to another, the cursor will move in the order that the controls are defined in the dialog regardless of where they are placed on the screen.

Using Controls to Write to a Data Set.

Controls can be used to enter data into a Blaise data set. This is demonstrated in the dialog *EnterCodes* which works in concert with the procedure *EnterCodesPro*c. These examples are found in the example setup *DIALPROC.MAN*. The setup must name the Blaise data model in the uses section and the Blaise data file must be declared as an updatefile in the file section.

```
USES
  NCS_EASY

 . . .

UPDATEFILE HHFile : NCS_EASY ('NCS_DATA', BLAISE)
```

The dialog *EnterCodes* has two controls that can enter data into a Blaise data set without invoking the Data Entry Program.

```
DIALOG EnterCodes "Enter management codes."
  HELP = 37
  SIZE     = (65, 12)
  DEFAULTBUTTON = NO
  VALIDATEDIRECT = YES
  CONTROL Manage.SendForm
    LABEL      = ('Send Form:  ', 2, 2)
    POSITION   = (28, 2)
    EDIT       = YES
    STATUS     = 'Set SendForm to Yes or No.'
  CONTROL Manage.ToNumber
    LABEL      = ('To interviewer number:  ', 2, 5)
    POSITION   = (28, 5)
    EDIT       = YES
    STATUS     = 'Number of the receiving interviewer'
  BUTTON SomeButtons
    CAPTION = '  ~O~K    '
    VALUE   = OK
    STORE = YES
```

The controls are for fields *SendForm* and *ToNumber* in a block called *Manage* in the Blaise data model.

The connection between the controls in the dialog and the Blaise Data set is made in *EnterCodesProc*.

```
PROCEDURE EnterCodesProc
  Region := EMPTY                        {AUXFIELD}
  Stratum := EMPTY                       {AUXFIELD}
  SampleNum := EMPTY                     {AUXFIELD}
  EnterID                                {DIALOG}
  HHFile.GET(Region,Stratum,SampleNum)
  IF HHFile.RESULTOK THEN
    EnterCodes                           {DIALOG}
    HHFile.WRITE
  ELSE
    DISPLAY('Invalid ID numbers, try again.', WAIT)
  ENDIF
ENDPROCEDURE
```

Three identification auxfields *Region*, *Stratum*, and *SampleNum* are set to empty in case they had obtained values from a previous dialog. Then the dialog *EnterID* gets current values for them. The *GET* statement points to a form in the data file corresponding to the values of these auxfields. *RESULTOK* ensures that a form in the data file was obtained (whether a correct ID number was entered). If so, the dialog *EnterCodes* is used to obtain values for *SendForm* and *ToNumber*. If not, a message is displayed to the user. The statement that permanently writes the data to the Blaise data file on disk is *HHFile.WRITE*.

3.6 Button Dialog Element

Buttons are almost always used with lookup and control elements and invoke a command (an action) when pressed. A button can be used to accept the values that are entered in control elements, exit the dialog without accepting any of the typed values, start a procedure, start another dialog, or start a search for a form within the file. The user may choose a form from the Blaise data set with lookup and use a button to invoke the Data Entry Program for that form.

Invoking a button

To invoke a button and its action, the user can <Tab> to it (or otherwise arrive there) and press <Enter>, click it with a mouse, or if the developer has provided a hot key for it, press <Alt ?> where ? stands for the hot key letter. In this manual, we use the phrase ‘press the button’ to stand for all of these actions.

Buttons can be associated with a button-related auxfield though this is not necessary. If the button is associated with an auxfield in the setup, and the button is pressed, then the auxfield gets the value of the button, (unless *STORE = NO*, see below).

Buttons and Auxfields

When you press a button, you can cause a value to be computed into an auxfield. It is the same as computing a value into an auxfield with an assignment like in a Manipula procedure, except in this case, the assignment is made interactively by the Maniplus user. If you assign a value to an auxfield, the setup can take further action based on the value of the auxfield. The definition of auxfields *OneOKButton* and *SomeButtons* in the example setup *DIALPROC.MAN* are as follows.

```
AUXFIELDS
  OneOKButton :
   (ok "Press this button for OK@!    ~O~K    ")

  SomeButtons :
   (ok "Press this button for OK@!  ~O~K   ",
    cancel "Press this button to Cancel@! ~C~ancel ",
    ready "Ready",
    dep "DEP",
    accept "Press this button to accept@! ~A~ccept ",
    insert "Insert",
    searchdia "Search dialog",
    dialg "Another dialog",
    aproc "Procedure")
```

The possible values of the button-related auxfields such as *SomeButtons* have no intrinsic meaning, they are merely suggestive of the action they could invoke. The function of a button's auxfield value is assigned in the button element itself. The auxfield *SomeButtons* has nine possible values. However, it is not necessary, and not usual, to use all nine values in any one dialog. The auxfield *SomeButtons* can be considered to be a generic button-related auxfield that can be used wherever its possible values are appropriate. In the example setup *DIALPROC.MAN* the dialog *ButtonsInDialog* makes use of five of the possible values of *SomeButtons*. On the other hand, several of the dialogs make use of the auxfield *SomeButtons.*

Default Status Line and Caption

In the definition of the auxfield *SomeButtons*, a few of the possibilities have a description that provide a default status line and Caption for any button that makes use of the possible value. For example:

```
SomeButtons :
   (OK "Press this button for OK@!  ~O~K   ",
```

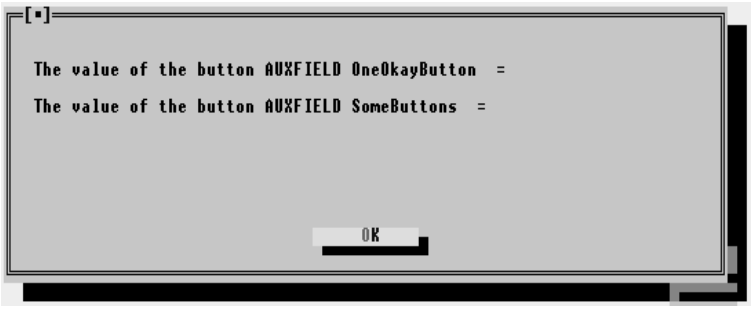
The default status line is the text before the symbol *@!*. This status line will appear in the status bar at the bottom of the screen. The default Caption is the text after the *@!*. This text will appear on the button itself. These default texts become part of the dialog unless they are overridden. They can be overridden with the properties status and caption when the button is defined.

Two or More Auxfields with Buttons

When a dialog with buttons is invoked, the auxfield associated with the buttons is set to empty awaiting the assignment of a value. Once the auxfield obtains a value from a button, the auxfield retains that value until another dialog making use of the auxfield is invoked. Sometimes you want the value of a button-related auxfield to be kept from one dialog to another. This can be accomplished by using two or more button-related auxfields. The setup *DIALOGS.MAN* has two such auxfields, *SomeButtons* and *OneOKButton*. Two of the dialogs in this setup show how to use two button-related auxfields in order to pass their values from one dialog to another, and how, when its dialog is invoked, the auxfield is set to empty once again. The example dialogs are *ShowSomeButtons* and *SubDialogWithButtons*.

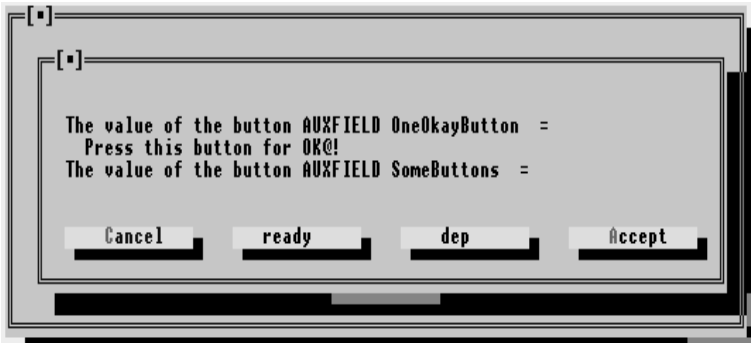
To see how they operate together, invoke *DIALOGS.MAN*, then choose the main menu choice *Buttons*, sub-menus *Sub dialogs* and then *Sub buttons*. The first dialog to appear has two lines of text displayed. The first shows the value of the auxfield *OneOKButton*. The second shows the value of *SomeButtons*. When the dialog is first invoked, the values of both auxfields are empty.

Figure 3.6.1. 'Sub buttons' dialog



Press the *OK* button. A second dialog comes up, showing the label of the auxfield *OneOKButton* whose value is *OK*.

Figure 3.6.2. The result after pressing OK



That is because the auxfield *OneOKButton* was associated with the first dialog. The second dialog is associated with the auxfield *SomeButtons*. Since *SomeButtons* is set to empty as the dialog is invoked, the dialog shows an empty value for it.

Now press the button *Accept*. The second dialog disappears, but the first one remains. This time however, it shows the value of *SomeButtons*, but still not the value of *OneOKButton*.

Figure 3.6.3. Back in the 'Sub buttons' dialog

You can experiment with different values (different buttons) in the second dialog to see what their effect is on the value of *SomeButtons*.

Defining a Button

A button is defined with the key word *BUTTON* followed by an optional *FieldID* which is an auxfield or an output field. It is usually followed by propertiess.

Two Button Examples

Two dialogs that show possibilities with buttons are *EnterData* in *DIALOGS.MAN* and *ButtonsInDialog in the setup DIALPROC.MAN*. *EnterData* is a dialog that combines controls with buttons. *ButtonsInDialog* combines a lookup with buttons using the properties *ONPRESS*. This latter example is covered under the lookup element discussion below.

Buttons in the Dialog EnterData

The buttons in the dialog *EnterData* are associated with controls. This example is especially designed to demonstrate the relationship between the properties *VALUE* and *STORE*. *VALUE* gives a value to the button-related auxfield or field. *STORE* determines if the values of the controls in the dialog should be stored (saved). A corresponding dialog, *DisplayData*, shows the values saved (if any) for the button-related auxfield *SomeButtons* and the controls.

The *OK* button is defined as follows.

```
BUTTON SomeButtons
  CAPTION = '    ~O~K    '
  STORE = YES
  STATUS = 'Save the entries but not the SomeButtons.'
```

In the *OK* button, there is no *VALUE* properties. When this button is pressed, the auxfield *SomeButtons* will still be *EMPTY*. However, the values of the controls are saved because of the properties *STORE = YES*. You can verify this by choosing sub-menu *Enter Some Data* under the main menu choice *Controls*. After entering data in the 12 controls, press the *OK* button. Then invoke the following sub-menu choice, *Display Some Data*. You will see that the values of the controls are displayed, but that the value of the auxfield *SomeButtons* is not. The value of this auxfield would be displayed in the bottom right corner next to the text ‘Button:’.

The *Accept* button is defined as follows.

```
BUTTON SomeButtons
  CAPTION = '   ~A~ccept   '
  VALUE = Accept
  STATUS = 'Save the entries and SomeButtons.'
```

The *Accept* button will store a value for the auxfield *SomeButtons* because the value setting is invoked. Specifically, the value stored will be *Accept*. The properties *STORE* is not seen here, but its default value is yes, when *VALUE* is set. Thus the values of the controls are stored. If you fill in the dialog *EnterData*, press the *Accept* button, then inspect the dialog *DisplayData* you will see both the values of the controls and the value of *SomeButtons* (look in the right bottom corner).

The *Ready* button is defined as follows.

```
BUTTON SomeButtons
  CAPTION = '   ~R~eady    '
  STATUS = 'Do not save the entries nor AUXFIELD
            SomeButtons.'
```

The *Ready* button will not store the auxfield *SomeButtons* and it will not store the value of any control, because no *VALUE* is set. If you enter data in *EnterData* and press the *Ready* button, you will see in *DisplayData* that no data were saved. Where no *VALUE* is set, *STORE = NO* by default.

The *Cancel* button is defined as follows.

```
BUTTON SomeButtons
  CAPTION = '   ~C~ancel   '
  VALUE   = Cancel
  STORE = NO
  STATUS = 'Do not save the entries nor AUXFIELD
            SomeButtons.'
```

The *Cancel* button shows that the property *STORE* controls the property *VALUE*. You might expect in this button that the value of the auxfield would be stored but not the values of the controls.

However, the value of the auxfield is not stored because *STORE = NO* takes precedence over the *VALUE* properties.

Summary of Relationship between Value and Store

The relationship between the propertiess value and store, and how they affect the contents of auxfields and fields represented by the button or controls is summarised in the following table. When *VALUE* is set, *STORE = YES* is the default, otherwise *STORE = NO* is the default. These defaults can be overridden.

	STORE = YES	**STORE = NO**
VALUE = YES	- Auxfield is saved. - Contents of controls are saved.	- Auxfield is not saved. - Contents of controls are not saved.
VALUE = NO	- Auxfield is not saved. - Contents of controls are saved.	- Auxfield is not saved. - Contents of controls are not saved.

Onpress Setting

The *ONPRESS* property of a button generates a command when a button is pressed. *ONPRESS* can invoke another dialog, a procedure (without parameters), or a search for a form in a file. If *ONPRESS* is not present for a button, then the button generates a command to close the dialog.

Onpress with a Control

ONPRESS is commonly used with controls. A simple example is taken from dialog *SubDialogWithButtons* in the setup *DIALOGS.MAN*.

```
DIALOG SubDialogWithButtons
 . . .
  BUTTON OneOKButton
     VALUE = OK
     ONPRESS = ShowSomeButtons   {DIALOG}
```

If the button with *VALUE = OK* is pressed, then the dialog *ShowSomeButtons* will be invoked.

The use of *ONPRESS = SEARCH* in conjunction with lookups, and *ONPRESS = ProcID*, are demonstrated further under *Lookup Dialogs Elements*.

Buttons Without Reference to Auxfields

Sometimes you may want to display a button in a dialog but not care if a button-related auxfield receives a value. This can be done by defining the button without a *FieldID* as shown.

```
BUTTON
    CAPTION = '   ~L~eave  '
```

A button like this could be used to allow the user an opportunity to leave the dialog without any trace of having been there. An example of this is shown in the *SeparatedLook* dialog in the next section, where the <Escape> key is disabled.

3.7 Lookup Dialog Element

A lookup invokes a data viewer on a file. The file can be an ASCII file, a temporary file or a Blaise data set. You can view one, all, or several fields of a file. The lookup might display a report, let a user select a form for further processing, or give a view on an ASCII file. Each field takes one column in the viewer. Only one lookup can be displayed in a dialog.

Since a lookup is a file viewing utility, it must be associated with a file declared in the file section of the setup.

The *USES* section describes two data models.

```
USES
  DATAMODEL MenuDescription   {Menu file description.}
    TYPE TFunctionKey =
      (ShiftF10 (40) "shiftf10",
       AltX (100) "altx")
    FIELDS
      MenuID : STRING[8]
      MenuText : STRING[20],   EMPTY
      FunctText : STRING[10],   EMPTY
      FunctKey : TFunctionKey, EMPTY
      StatusText : STRING[40],   EMPTY
      HelpID : 1..900,       EMPTY
      Program : STRING[10],   EMPTY
      RunParam : STRING[10],   EMPTY
  ENDMODEL

  DATAMODEL OneLine
    FIELDS
      OneLine : STRING[110]
  ENDMODEL
```

The first data model description is for the fields in the menu file. For this example setup, the menu file is the fixed file *AscMenu.Asc* with each field starting in a specified column. A version of this file is *AscMenu.Sep*. It contains the same information as *AscMenu.Asc* but the fields are separated by #.

The second *USES* file description is used to browse ASCII files. There is only one field. It represents the first 110 characters of each line of a file. In the file naming section of the setup, there are three *INPUTFILE* declarations. The first two refer to the file description of the menu file *MenuDescription*. The first is for viewing the fixed field version of the menu file, the second, for viewing the separated version of this file.

```
INPUTFILE
  MyMenu : MenuDescription ('AscMenu.ASC', ASCII)

INPUTFILE
  SeparatedFile : MenuDescription ('AscMenu.SEP', ASCII)
  SETTINGS
    SEPARATOR = '#'
```

In both of these, the file names are known in advance, so they are named here. In other words, the file names are hard-coded.

The following *INPUTFILE* statement does not name a file. This description will be applied to unknown files that are determined at a later point in the setup, or more dynamically, by the user during runtime. The setting *OPEN = NO* informs Maniplus that the developer takes care of opening and closing this file.

```
INPUTFILE AnyASCIIFile : OneLine
  SETTINGS
    OPEN = NO
```

The setup *DIALOGS.MAN* includes lookup dialogs *FixedLook*, *SeparatedLook*, *AnyASCIILook*, *SubsetOfFile*, and *DynaLook*. The following table indicates the dialog and its association with a file identifier.

DIALOG	USES (description)	FILE NAME
FixedLook	*MenuDescription*	*AscMenu.Asc*
SeparatedLook	*MenuDescription*	*AscMenu.Sep*
AnyASCIILook	*OneLine*	*Dialogs.Man*
SubsetOfFile	*MenuDescription*	*AscMenu.Sep*
DynaLook	*OneLine*	*determined by user*

The code for the lookup dialog *FixedLook* is shown.

```
DIALOG FixedLook
  HELP = 21
  LOOKUP MyMenu                          {file AscMenu.Asc}
  BUTTON OneOKButton
    VALUE = OK
    CAPTION = '    ~O~K    '
```

The lookup dialog *FixedLook* is invoked by the instruction *LOOKUP MyMenu*. This two word phrase associates the data viewer with the file represented by *MyMenu* which is *AscMenu.Asc*. This association of *MyMenu* and *AscMenu.Asc* was done already in the file section. The button allows the user to leave the lookup dialog as does the <Escape> key.

The code for the lookup *SeparatedLook* is shown. This allows the user to browse an *ASCII* separated file as if it were a fixed field file.

```
DIALOG SeparatedLook
  HELP = 22
  ESCAPE = NO
  LOOKUP SeparatedFile                   {file AscMenu.Sep}
    HEADER = NO
  BUTTON
    CAPTION = '    ~O~K    '
```

In this dialog, the <Escape> key is not allowed. The user has to press the button in order to leave the dialog. The button is not associated with any auxfield, thus no auxfield will get the value of the button. The identifier *SeparatedFile* was already

associated with the file *AscMenu.Sep* in the file section. *HEADER = NO* means that no header will appear atop the lookup table.

In the following lookup the identifier *AnyASCIIFile* does not have a pre-determined association with a file in the file section.

```
DIALOG AnyASCIILook
  HELP = 23
  LOOKUP AnyASCIIFile     {no pre-set file association}
  BUTTON
    CAPTION = '    ~O~K    '
```

However, the association to a file is made in the case structure later in the setup.

```
CASE program OF
 . . .
  'ASCIIlook'  : Name := 'Dialogs.Man'
                 AnyASCIIFile.OPEN(Name)
                 AnyASCIILook('Browse file DIALOGS.MAN')
                 Name := EMPTY
```

The file associated with *AnyASCIIFile* is named in the case structure to be *Dialogs.Man*. When the setup is invoked, the user will see the contents of this file and not be able to choose another file to view. The auxfield *Name* is used in this case to hard-code the file name. It is used in a further example to allow the user to choose which file to browse.

Viewing a Subset of the Fields of a File

For a large data model, or an ASCII file with many fields, the user would not need to see the contents of the whole data set. It is possible to view a smaller part of the file with the *FIELDS* properties. This is shown with dialog *SubsetOfFile*. It shows just two fields of the separated version of the menu file.

```
DIALOG SubsetOfFile
  HELP = 24
  LOOKUP SeparatedFile
    POSITION = (15, 3)
    SIZE = (50, 7)
    STATUS = 'View a few fields of the file ASCMENU.SEP'
    SEPARATOR = YES
    HELP = 901
    FIELDS
      MenuID "Menu ID"
      MenuText "Menu text"
  BUTTON
    CAPTION = '    ~O~K    '
```

Vertical Lines in the Lookup Viewer

The two fields in the lookup are separated by a vertical line due to the property *SEPARATOR = YES*, not because this is a view on a separated file.

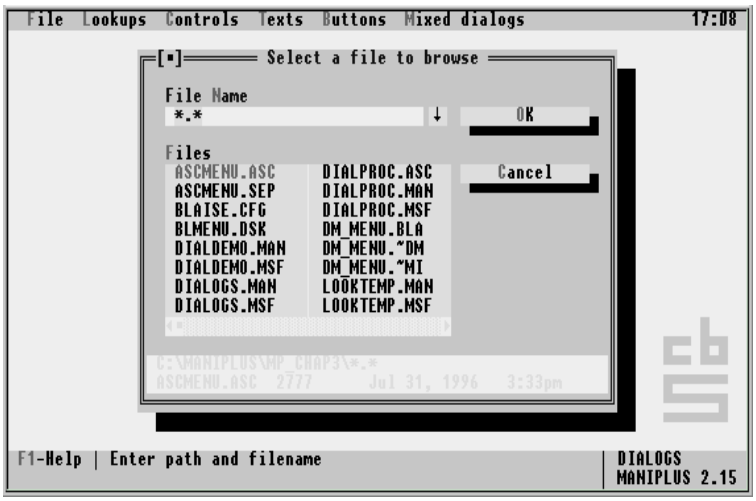
The User Chooses a File to Browse

It is possible to allow the user to choose which file to browse. This is done by using the pre-defined dialog which is generated from the selectfile command. All commands to make this work are in the following CASE structure.

```
CASE program OF
 . . .
  'dynalook' : Name:= SELECTFILE('Select a file to ' +
                                 'browse', Name, '*.*')
               IF Name <> EMPTY THEN
                 AnyASCIIFile.OPEN(Name)
                 AnyASCIILook('Browse file ' + Name)
               ENDIF
               Name := EMPTY
```

The auxfield *Name* is assigned when the user selects a file from the selectfile dialog shown below.

Figure 3.7.1.
Select file dialog



The *SELECTFILE* command in this example lets the user select any file to browse with the *.* parameter and places the chosen file name into the auxfield *Name*. It is possible to constrain users to files with particular names using wild card place holders. The reader is referred to the Reference Manual for further details on *SELECTFILE*. After the file to browse is chosen, the user-defined lookup dialog *AnyASCIILook* is called to browse the file. Any file in any directory can be viewed with this example, not just ASCII files. If users chooses a binary file, such as an executable file, to browse, they may see many funny characters in the viewer. The lookup dialog views the file in read-only mode and thus cannot do any harm to it.

Onpress = ProcID with Lookup

The following three code boxes demonstrates the use of onpress with a lookup dialog. This example code is taken from the setup *DIALPROC.MAN*, dialog *ButtonsInDialog*. The lookup element is defined first.

```
DIALOG ButtonsInDialog 'Selected fields from data file.'
 . . .
  LOOKUP HHFile
    SIZE = (75, 11)
    STATUS = 'A view on some of the data in the file'
    POSITION = (2, 1)
    SEPARATOR = YES
    FIELDS Ident.SampleNum Manage.FormStat Address.Town
           Address.Street Manage.SendForm
           Manage.ToNumber Person.SurName
```

The lookup allows the user to browse the forms in the Blaise data set, viewing just those fields deemed to be important for survey management. As the user scrolls vertically through the table, a file pointer is set to the record in the data file that corresponds to the line of the lookup the cursor is on. That is, if users want to choose a form for further processing, they merely have to place the cursor on the line in the table that represents the form. Depending on how the dialog is programmed, they may be able to invoke the Data Entry Program by pressing <Enter>, or by pressing a button provided for that purpose.

The following button will start the Data Entry Program, by calling the procedure *GetFormProc*.

```
  BUTTON SomeButtons
    CAPTION = ' start ~D~ep '
    VALUE = dep
    POSITION = (17, 19)
    ONPRESS = GetFormProc             {PROCEDURE}
    STATUS = 'Start DEP for currently selected form.'
```

When the user presses *Start DEP*, the *ONPRESS* command calls procedure *GetFormProc*.

```
PROCEDURE GetFormProc
  HHID := STR(Ident.Region) + ';' + STR(Ident.Stratum) +
          ';' + STR(Ident.SampleNum)
  IF CONFIRM('Invoke instrument for ' +
    '@/ Region: ' + STR(Ident.Region) +
    '@/ Stratum: ' + STR(Ident.Stratum) +
    '@/ SampleNum: ' + STR(Ident.SampleNum) +
    '@/ Town: ' + Address.Town +
    '@/ Street: ' + Address.Street +
    '@/ Name: ' + Person.SurName) THEN
    Reslt := HHFile.EDIT('/G /X /FNCS_DATA /K' + HHID)
  ENDIF
ENDPROCEDURE
```

The procedure calculates the value of auxfield *HHID* from the primary keys of the data model *Ident.Region*, *Ident.Stratum*, and *Ident.SampleNum*. The values of these primary keys are taken from the form in the data file corresponding to the position of the cursor in the lookup. A confirm instruction is generated giving users an opportunity to verify the values of the primary keys and also the town, street, and surname of the household. They can invoke the Data Entry Program by pressing <Enter> or can avoid invoking it at this point, if the wrong form is chosen, by pressing <Escape>. If users pull up the wrong form, they can leave the Data Entry Program by pressing <Alt - X>.

The *EDIT* method in the line:

```
Reslt := HHFile.EDIT('/G /X /FNCS_DATA /K' + HHID)
```

calls the Data Entry Program (DEP). The *EDIT* method and *EDIT* function are covered more thoroughly in Chapter 4.

Onpress = Search
Searching a Data File for a Particular Form

The lookup dialog is very powerful. Users can scroll up, down, left, and right with arrow and paging keys. However, if the number of forms in the data file is large (say over 200), they may need other ways to find an appropriate form. One way is to search a file with a key field, whether the key is primary or secondary. This example is taken from *DIALPROC.MAN*, dialog *ButtonsInDialog*.

Primary and Secondary Keys

This example assumes knowledge of primary and secondary keys. They are documented in full in Chapter 3 of the Developer's Guide. A primary key uniquely identifies a form. There can be only one primary key, but it can be composed of two or more fields. The primary key for the data model *NCS_EASY* is *Ident*, which is a block composed of the fields *Region*, *Stratum*, and *SampleNum*. Thus this is a three part primary key.

```
PRIMARY
  Ident  {Block of 3 FIELDS, Region, Stratum, SampleNum}

SECONDARY
  TownOfHome = Address.Town
  StreetName = Address.Street
  StatusOfForm = Manage.FormStat
  SampleNumber = Ident.SampleNum
```

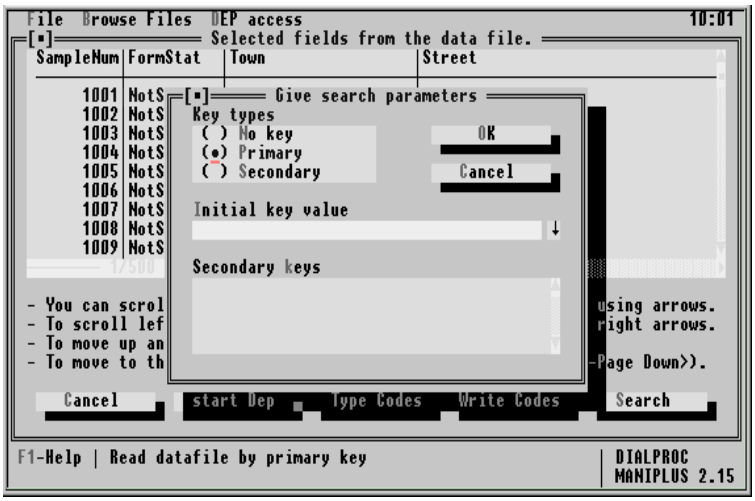
Secondary keys are useful when you wish to invoke or process a particular kind of form, say all forms in a certain town. In the example data model *NCS_EASY.BLA*, there are four secondary keys. When browsing in the form search dialog, the user will be able to choose between the primary key and the secondary keys when searching for a specific form. Note that the field *SampleNum* is both part of the primary key and a secondary key by itself.

The onpress below invokes a search on a file.

```
  BUTTON SomeButtons
    CAPTION = '  ~S~earch   '
    VALUE = searchdia
    ONPRESS = SEARCH(HHFile)
    POSITION = (62, 19)
    STATUS = 'SEARCH DIALOG to search for form.'
```

When the search button is pressed, the search dialog is invoked.

Figure 3.7.2.
The Search dialog



Users can choose which kind of key to search on, *NoKey*, *Primary Key*, or *Secondary Key*.

If they choose *Primary Key* they can type in the whole primary key number, with no spaces between its components, or with a semicolon in between. In other words to type in *Region 10*, *Stratum 2001*, and *SampleNum 1001*, either 1020011001 or 10;2001;1001 can be typed. Since the original lookup dialog is sorted in order of primary key, users would be better off using that to search for a particular form by primary key.

The search dialog is most powerful if it is used with secondary keys. Once a user has indicated a search on secondary keys, a particular secondary key must be chosen. For example, to search on the street name, the user can <Tab> to the secondary key menu, choose the key *StreetName*, <Shift-Tab> back to the rectangle, then type in a name of a street. In this generated data setup, streets have names such as *K-STREET*. If a user searches for this name, and it exists, then the cursor will land on the first instance of a record with *K-STREET*. If no such record exists, the cursor will land on the nearest record, for example the last record with a *J-STREET* address.

To search on a form number which represents the order in which it was entered, use *NoKey*. To get the 24th entered form, the user just enters the number 24 after choosing *NoKey*. For an interviewer, this option would be rarely used.

Write Information to a Blaise Data File

To write information to a Blaise data file directly from Maniplus, a WRITE statement must be used. The following button definition calls a procedure *WriteCodesProc*.

```
BUTTON SomeButtons
  HIDE = YES
  READY = YES
  CAPTION = '~W~rite Codes'
  VALUE = aproc
  ONPRESS = WriteCodesProc          {PROCEDURE}
```

When the button is pressed, the underlying lookup will disappear because of the property *HIDE = YES*. When the procedure that is called is finished, the dialog is closed and the user is returned to the menu because of the property *READY = YES*.

The procedure *WriteCodesProc* calls the dialog *EnterCodes* and writes the information to file.

```
PROCEDURE WriteCodesProc
  HHFile.GET(Ident.Region, Ident.Stratum,
             Ident.SampleNum)
  IF HHFile.ResultOK THEN
    EnterCodes        {DIALOG}
  ELSE
    DISPLAY('Failed to GET form.', WAIT)
  ENDIF
  HHFile.WRITE
ENDPROCEDURE
```

This procedure does not invoke the Data Entry Program (DEP). Rather, it writes information to the Blaise data file directly. In this case, the get statement puts a file pointer at the form whose primary keys correspond to those in the lookup where the cursor is sitting. The dialog *EnterCodes* is invoked, and this is where the user enters the code numbers. When the dialog is terminated,

the write instruction ensures that the entered data are written to file on disk.

3.8 More Powerful Lookups with TEMPORARYFILE

Some of the power of lookup dialogs has already been demonstrated. Lookups can be made even more powerful if the user has the ability to sort the table on any column and/or if the lookup table presents a selected subset of the forms in the Blaise data set. When a lookup is used that can be sorted and can hold a subset of forms, it is possible to combine the functionality of many lookup dialogs into one.

In order to provide this power to the user, a temporary file is displayed in the lookup dialog. The temporary file has connections to the Blaise data set through its primary keys. When users choose a form in the lookup, (they are choosing a form from the temporary file), they are connected to the corresponding form from the Blaise data set.

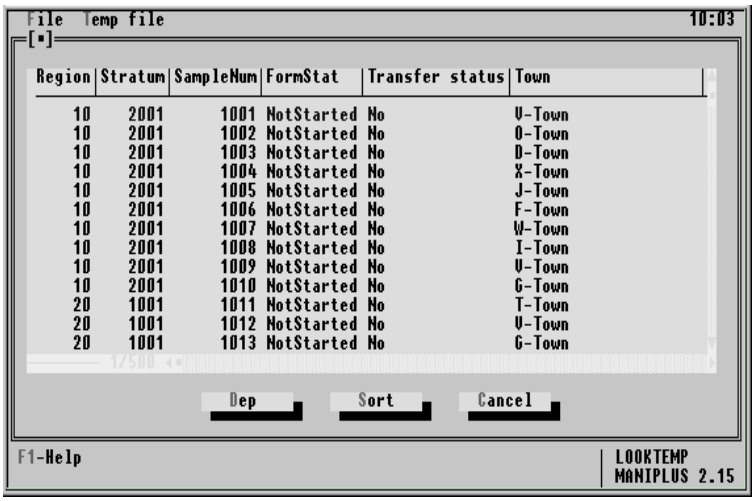
Two example lookup dialogs

The example setup *LOOKTEMP.MAN* holds two lookup dialogs that demonstrate how to enhance the power of lookups by using a temporary file. They are both found under the main menu heading *Temp file*. The first example demonstrates sorting possibilities. It is accessed by the sub-menu choice *List the whole file*. The second example combines sorting with subsetting and in fact is an extension of the first. It is accessed with the sub-menu choice *List a partial file*.

List the whole file

The lookup displayed as the result of choosing the sub-menu choice *List the whole file* is shown below.

Figure 3.8.1. 'List the whole file' dialog



The example data shown are generated from another Manipula program (*NCS_DATA.MAN)* and thus the entries you see on your screen will be similar to, but not exactly the same as, the data generated for this example. The table is large but the dialog does not display all of the data. The data are both wider and deeper than the lookup window.

Navigation in the lookup

The user can navigate several different ways as described by the bullets below. If necessary, the table will scroll.

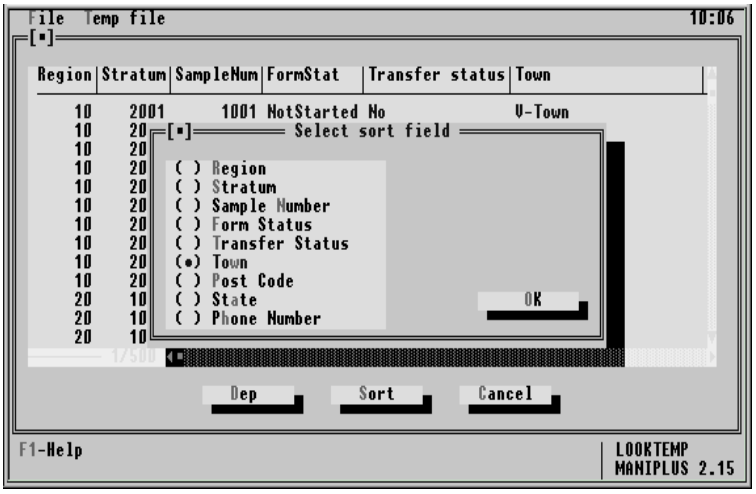
- The arrow keys (up, down, right, and left) navigate in the direction indicated. This is the slowest way to navigate.
- To navigate vertically a page at a time, use <Page Down> or <Page Up>.
- To get quickly to the top form, <Ctrl Page Up>.
- To get quickly to the bottom form, <Ctrl Page Down>.
- To navigate horizontally one column at a time, use <Ctrl right arrow> or <Ctrl left arrow>.

Sort button

It is the *Sort* button that separates this lookup dialog from other examples mentioned so far. This example allows the user to

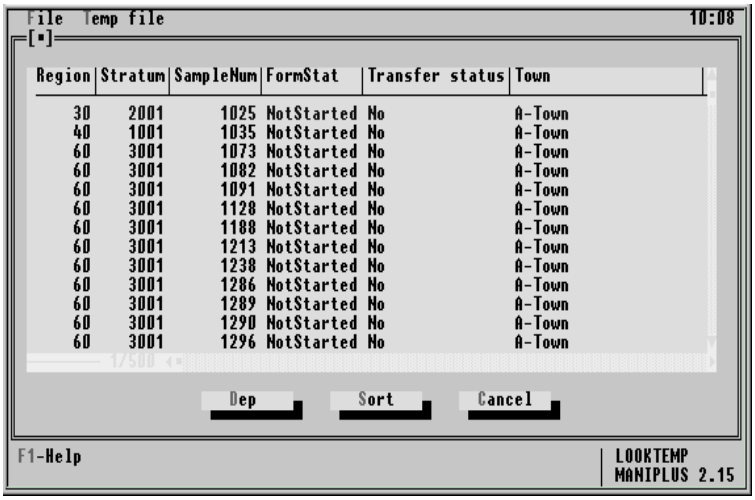
sort the table on any of the columns. When the *Sort* button is pressed a control dialog pops up. In the example below, the user has chosen the *Town* to sort the table.

Figure 3.8.2. 'Select sort field' dialog



After the town has been chosen as the sort column, the table is instantly redisplayed.

Figure 3.8.3. Lookup after sort



You can see above that the table now displays the forms in the order of the towns.

Entering DEP

The user can enter the data entry program from the lookup dialog either by pressing <Enter> or by pressing the *DEP* button. The form that is invoked is the one corresponding to the position of the cursor on the lookup table.

Re-sort

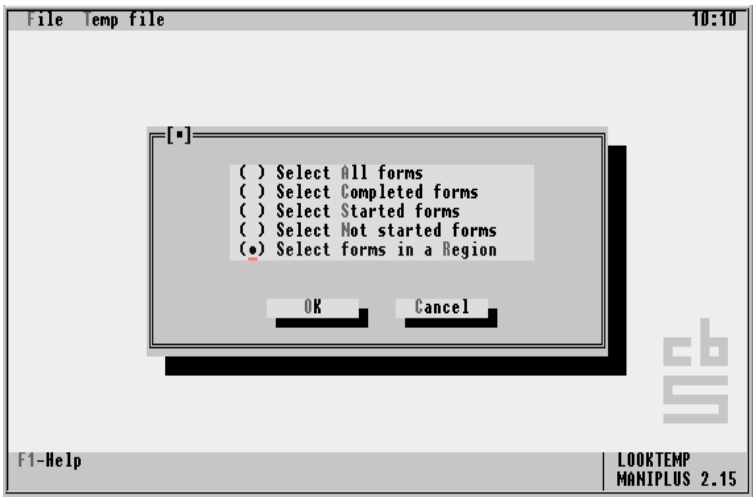
The lookup table is now sorted by *Town*. In *DEP*, the user is free to change the field *Town* (at least in this example, but you may not allow this for some fields). If the town name is changed, then when *DEP* is left and the lookup table is redisplayed, the form with the changed town name is re-sorted. So if the field *Town* is changed from *A-Town* to *Z-Town* the location of the form in the table will move from the top of the table to the bottom, as long as the sort order is *Town*. However the cursor stays with the form that was just changed in *DEP*. This is how the user knows which form was just processed.

List a partial file

The lookup dialog accessed with the sub-menu choice *List a partial file* is an extension of the previous one. It has all the capability of the former, but gives the user the ability to choose which forms to see in the table. For example, if interviewers wants to work in a particular region, they can list just those forms in that region, leaving others out of the table.

When the sub-menu choice *List a partial file* is invoked, a control dialog pops up asking the user which part of the file to view.

Figure 3.8.4.
'Select forms' dialog

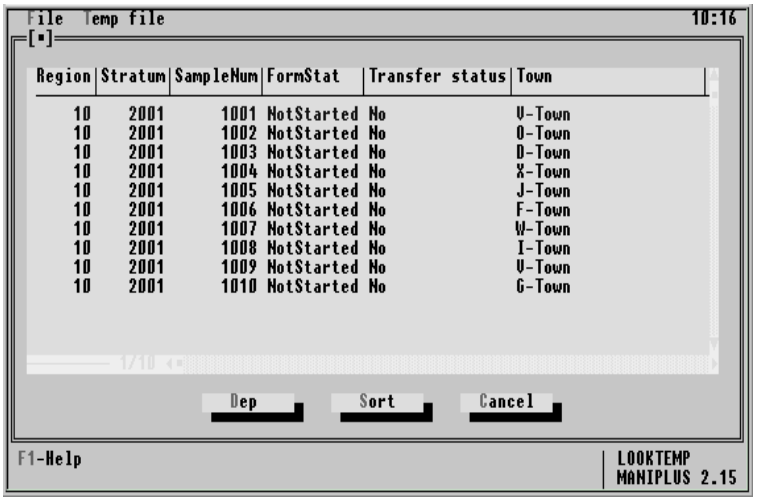


Users can choose any of five choices. The first choice *Select all forms* will give them the same table in the previous example. Three of the choices are by form status. They can look at *Completed* forms, *Started* forms, or *Not Started forms*. Finally the user can choose *Region*. If a user chooses *Region* another dialog pops up asking for a *Region* number.

Figure 3.8.5.
Specify Region number

In this example, if 50 example forms were generated, the valid *Regions* are 10, 20, 30, 40, or 50. If the user chooses *Region* 10, then only those forms are displayed.

Figure 3.8.6. Lookup with selected forms for region 10



The user still has the ability to sort the forms as in the first example.

Developing a lookup that can sort

The developer must execute several steps to give the user the kind of power and flexibility demonstrated here. These steps are summarised here and in the programmer comments written in the example code for the example *LOOKTEMP.MAN*.

Describe and name the temporary file

The structure of the temporary file, like any other file in a Manipula or Maniplus setup is described in the uses section and named in the file section. The essential parts are displayed below.

```
{USES section}
DATAMODEL TempFile
 . . .
  PRIMARY       {Same primary keys as Blaise data file.}
    Region,
    Stratum,
    SampleNum

  SECONDARY  {Re-defined as needed for different sorts.}
    SortField
 . . .
  FIELDS        {All but one from the Blaise data file.}
    Region  : Ttwo9s
    Stratum  : Tfour9s
    SampleNum : 1000..9000
    FormStat : Form_Status
    SendForm : TYesNo
    Town : STRING[20]
    State : STRING[20]
    PostCode : STRING[10]
    PhoneNum : STRING[12]
    SortField : STRING[20]  {Not from Blaise data file.}
ENDMODEL
```

Only the fields needed for the lookup are brought into the temporary file. The field *SortField* is a generic sort field. It will be defined and re-defined every time the user wants to sort by a different (column) field. By making it a *STRING*, the table will be able to sort by text values, such as town name, as well as by numeric or enumerated types. The file section is as follows.

```
TEMPORARYFILE TempTable : TempFile
  SETTINGS
    KEY = SECONDARY
```

The temporary file will be known to the rest of the setup as *TempTable*. It has the description (shown above) called *TempFile*. The setting *KEY = SECONDARY* ensures that the table will be displayed in the order of the secondary key, in this case, *SortField*. In fact, the sorting is carried out by redefining the values of *SortField*. You won't see any sort instruction in this setup.

Filling the temporary table

When the appropriate menu choice is pressed, the temporary file must be filled from the Blaise data file. There are two procedures that accomplish this. The first, *FillTempData*, is used to fill one line of the temporary table.

```
PROCEDURE FillTempData
   TempTable.INITRECORD
   TempTable.Region := Ident.Region
   TempTable.Stratum := Ident.Stratum
   TempTable.SampleNum := Ident.SampleNum
   TempTable.FormStat := Manage.FormStat
   TempTable.SendForm := Manage.SendForm
   IF Address.Town <> EMPTY THEN
     TempTable.Town := Address.Town
   ELSE
     TempTable.Town := '·'
   ENDIF
{etc.}
ENDPROCEDURE
```

There are not any file references (to *HHFile1* or *HHFile2*) in this procedure. This is taken care of by the calling procedure.

A sly trick is employed when the value of *Address.Town* is empty. Instead of leaving *TempTable.Town* empty in this case, a small dot (not a period) is placed in the field. When the Blaise data set (hence the temporary file) is partially filled in, and some records have values for Town and others do not, the sort on town will put the records with valid values at the top of the list. This is because the small dot has the ASCII value of 250, which is higher than any number or letter in the alphabet. You get the small dot when you press <Ctrl Space Bar> in the text editor.

You can also use the *CHAR* function.

```
ELSE
   TempTable.Town := CHAR(250)
```

The procedure, *FillTempTable*, calls *FillTempData* as many times as needed in order to fill in all the lines of the table (file).

```
PROCEDURE FillTempTable
  HHFile2.OPEN
  HHFile2.READNEXT
  WHILE (HHFile2.RESULTOK) DO
    FillTempData                    {PROCEDURE}
    TempTable.WRITE
    HHFile2.READNEXT
  ENDWHILE
  HHFile2.CLOSE
ENDPROCEDURE
```

The developer takes care of opening and closing the Blaise data file. The updatefile section *HHFile2* is used because without the setting *ACCESS = SHARED* and with the sub-section *FILTER* the data transfer from the Blaise data file to the temporary file is vastly accelerated (see below).

It would be possible to write one procedure to do what these two procedures do. However, there are times when we will want to fill just one line of the table, not all lines at one time.

Define references to the Blaise data file

The uses section needs a description of the Blaise data file which is done by referring to the Blaise data model *NCS_EASY.BLA*.

```
USES
  NCS_EASY
```

An updatefile and inputfile sections are defined for the Blaise data file.

```
UPDATEFILE HHFile1 : NCS_EASY ('NCS_DATA', BLAISE3)
  SETTINGS
    OPEN = NO
    ACCESS = SHARED

INPUTFILE HHFile2 : NCS_EASY ('NCS_DATA', BLAISE3)
  SETTINGS
    OPEN = NO
  FILTER
    Ident
    Manage
    Address
```

Both file sections refer to the same data file. Technically speaking, this example could have been written with just one *UPDATEFILE* section. However, many applications will make use of an *UPDATEFILE* and an *INPUTFILE* sections when there is a temporary file involved. Since both file sections refer to the same data file, the setting *OPEN = NO* is necessary. The developer will have to take care of opening and closing files in the appropriate places in the procedures (see below).

Performance and data integrity notes

The first updatefile section will be used to invoke the Data Entry Program (DEP). It has the setting *ACCESS = SHARED*. This setting will help to protect the data file in case the computer fails. However, the setting *ACCESS = SHARED* slows down file access greatly.

When filling the temporary file from the Blaise data file, speed may be of the essence. So the inputfile section does not have the *ACCESS = SHARED* setting. It does have a *FILTER* subsection, naming just those blocks that contain the data needed from the Blaise data file. Only those blocks from the Blaise data file are brought into memory when data are transferred from one the Blaise data file to the temporary file.

Two file sections for one file

The use of two file sections to refer to one file generalises to other situations, especially where Manipula is called from a Maniplus setup. If a Manipula program generates a report from the management block, you would not want to bring all of a large data file into memory just to obtain values for a few fields per record.

Auxfield for sorting

The auxfield for sorting in this example is called *SortAUXField*.

```
SortAUXField : (TheRegion "~R~egion",
                ItsStratum "~S~tratum",
                ASampleNum "Sample ~N~umber",
                TheFormStat "~F~orm Status",
                DoSendForm "~T~ransfer Status",
                ATown "To~w~n",
                APostCode "~P~ost Code",
                AState "St~a~te",
                APhoneNum "P~h~one Number")
```

It gets appropriate user-chosen values from a dialog (described below). In this example, the user has nine sort choices.

Filling the SortField

The field *SortField* from the temporary file is a generic sort field that will get its values depending on the sort order that the user wants. This is done in the procedure *FillSortField*.

```
PROCEDURE FillSortField
  CASE SortAUXField OF
    TheRegion : TempTable.SortField :=
                      STR(TempTable.Region)
    ItsStratum : TempTable.SortField :=
                      STR(TempTable.Stratum)
    ASampleNum : TempTable.SortField :=
                      STR(TempTable.SampleNum)

    {A trick!}
    TheFormStat :
         IF TempTable.FormStat = Started THEN
           TempTable.SortField := '1'
         ELSEIF TempTable.FormStat = NotStarted THEN
           TempTable.SortField := '2'
         ELSEIF TempTable.FormStat = Complete THEN
           TempTable.SortField := '3'
         ENDIF

    DoSendForm : TempTable.SortField :=
                      STR(TempTable.SendForm)
    ATown : TempTable.SortField := TempTable.Town
    AState : TempTable.SortField := TempTable.State
    APostCode : TempTable.SortField :=
                      TempTable.PostCode
    APhoneNum : TempTable.SortField :=
                      TempTable.PhoneNum
  ENDCASE
ENDPROCEDURE
```

The user chooses a value of the auxfield *SortAUXField* in a control dialog. If *TheRegion* is chosen, the instructions to the

right of the ':' are carried out, namely to fill *SortField* with the value of *Region*.

By defining and redefining the secondary key *SortField*, as the user chooses new sort options, the sort is executed. Secondary keys are sorted automatically in a Blaise data file. This is how they are useful and how they allow different kinds of access to the data sets. The sort on a secondary key is always in ascending order. This is usually what you want, however you can change the sort order on a field if you wish. For example, the field *FormStat* has three values, *NotStarted*, *Started*, and *Complete*, with values 1, 2, and 3 respectively. This may not be the optimal order for interviewers to see them. They may prefer to have the completed forms sorted last, the started ones sorted first, and the not started ones in the middle. To accommodate these other sort preferences you can employ the appropriate code in the *CASE* statement as shown above.

The dialog definition that allows the user to choose a sort field is shown.

```
DIALOG SelectSortField "Select sort field"
  SIZE = (50,12)
  CONTROL SortAUXField
    POSITION = (10,2)
    LABEL = NO
  BUTTON OneOKButton
    VALUE = OK
```

The *CONTROL SortAUXField* gives a value to the auxfield that controls the sort and which was listed above. The button *OneOKButton* is given the value *OK*. The procedure *SortTempTable*, shown below, calls the dialog above. It then executes file procedures needed to refill *SortField* calling the *FillSortField* procedure many times.

```
PROCEDURE SortTempTable
  SelectSortField                              {DIALOG}
  IF OneOKButton = OK THEN
    TempTable.RESET
    TempTable.SETREADKEY(PRIMARY)
    FOR Count:= 1 TO TempTable.FORMCOUNT DO
      TempTable.READNEXT
      FillSortField                            {PROCEDURE}
      TempTable.WRITE
    ENDDO
    TempTable.SETREADKEY(SECONDARY)
  ENDIF
ENDPROCEDURE
```

The *SETREADKEY(PRIMARY)* causes the *FOR - DO* loop to proceed through the temporary file in the order of the primary key. This ensures that every form in the data file is processed, no matter what secondary key is chosen at the moment. After all forms are processed, the secondary key is designated again as the key to control the processing order.

Invoking and showing the lookup table

The dynamic lookup table is invoked in a lookup dialog, like those documented above.

```
DIALOG ShowTempTable
  LOOKUP TempTable
    FIELDS
      Region
      Stratum
      SampleNum
      FormStat
      SendForm "Transfer status"
      Town
      State
      PostCode
      PhoneNum
  BUTTON SomeButtons
    VALUE = dep
    CAPTION = '  ~D~ep    '
    ONPRESS = EditEntry
  BUTTON SomeButtons
    VALUE = SortB
    CAPTION = '  ~S~ort   '
    ONPRESS = SortTempTable
  BUTTON SomeButtons
    VALUE = Cancel
    CAPTION = ' ~C~ancel  '
```

The fields property is used because not all fields in the temporary file need to be listed. Namely, *SortField* is not displayed as its necessary role takes place in the background. If you were having trouble debugging the sorting, then you could bring *SortField* on the screen temporarily. Otherwise, there is no use for it on the screen and it would only take up valuable space. For *SendForm* a column header 'Transfer status' is implemented because *SendForm* is not clear to the user.

The procedure *DynaLookup* calls this lookup dialog.

```
PROCEDURE DynaLookup
  SortField := STR(Ident.Region) {Beginning sort value.}
  FillTempTable                  {PROCEDURE}
  TempTable.RESET                {Pointer on 1st form.}
  ShowTempTable                  {Lookup DIALOG}
ENDPROCEDURE
```

To avoid confusion, *SortField* is given a starting value of *Region*. The procedure *FillTempTable* is invoked to transfer data from the Blaise data set to the temporary file. The instruction *RESET* puts the pointer (the cursor) on the first line of the table. Then the dialog *ShowTempTable* is invoked. Here is where the user can invoke different sorts on the lookup table and invoke *DEP*.

Invoking DEP

Invoking *DEP* from this dynamic lookup table is done just as it is from any other lookup table as described above. This is done in the procedure *EditEntry*.

```
PROCEDURE EditEntry
  HHFile1.OPEN
  Reslt := HHFile1.EDIT( '/G /K' + STR(TempTable.Region)
           + ';' + STR(TempTable.Stratum)
           + ';' + STR(TempTable.SampleNum) + ' /X ');
  HHFile1.CLOSE
  FillTempData                       {For one record.}
  TempTable.WRITE
ENDPROCEDURE
```

After *DEP* is invoked and then exited, the procedure *EditEntry* calls the procedure *FillTempData* which updates the temporary

file (hence the lookup). This ensures that if the value of the sort field is changed for the form, that the form will be re-sorted to its proper place in the lookup, and that all of its values are updated.

After leaving *DEP*, the last form edited is still present in *HHFile1*. That is, all fields in that form are ready to be accessed without performing a read.

Subset of forms

In addition to sorting forms in a lookup dialog, it is often desired to display just a subset of forms. For example the user may want to look at forms from just a region or only those that are started and need to be completed. The example sub-menu choice for this possibility is *List a partial file*.

An auxfield that represents the possible subsetting possibilities is needed. The auxfield for subsetting of forms in this example is called *SubsetAUXField*.

```
SubsetAUXField : (All "Select ~A~ll forms",
                  Complete "Select ~C~ompleted forms",
                  Started "Select ~S~tarted forms",
                  NotStarted "Select ~N~ot started
forms",
                  ARegion "Select forms in a ~R~egion"),
                  EMPTY
```

The possible choices in this auxfield are displayed in the control dialog *ChooseFormType*. If *ARegion* is chosen then another control dialog *WhichRegion* is invoked to obtain the region number. These dialogs can be viewed in the source code for the example setup *LOOKTEMP.MAN*. Once a choice is made, the procedure *FillPartialTable* is invoked to clear out the temporary file (if it exists already) and to rebuild it, with just a subset of forms. The first part of the procedure is shown.

```
PROCEDURE FillPartialTable
  TempTable.ERASE
  SubsetAUXField := EMPTY
  ChooseFormType                          {DIALOG}
  Region := EMPTY
  IF SubsetAUXField = ARegion THEN
    WhichRegion                           {DIALOG}
  ENDIF
```

The auxfield *SubsetAUXField* is set to empty in case it had a value from a previous sort. Once the user chooses a value to subset by in the dialog *ChooseFormType* the procedure opens the Blaise data file using updatefile *HHFile2* to increase performance (see discussion above). Then in a series of IF - *ELSEIF* statements, the procedure fills the temporary file with the appropriate records, calling procedure *FillTempData*. The procedure *FillPartialTable* continues. Only some of the lines are displayed.

```
  IF SubsetAUXField <> EMPTY THEN
    HHFile2.OPEN
    HHFile2.RESET               {Pointer on 1st record.}
    WHILE (HHFile2.RESULTOK) DO
      IF SubsetAUXField = ARegion THEN
        IF Ident.Region = Region THEN
          FillTempData                  {PROCEDURE}
        TempTable.WRITE
        ENDIF
      ELSEIF SubsetAUXField = All THEN
        FillTempData                    {PROCEDURE}
        TempTable.WRITE

      {More ELSEIF lines.}

      ENDIF
      HHFile2.READNEXT           {Go to the next form.}
    ENDWHILE                     {Stop after last form.}
    HHFile2.CLOSE
  ENDIF {SubsetAUXField <> EMPTY}
ENDPROCEDURE
```

The *WHILE - ENDWHILE* structure ensures that all forms of the Blaise data file are processed. *RESET* places the file pointer at the top of the file and *READNEXT* moves the pointer from one record to the next.

Finally the partial lookup table is invoked in the procedure *PartialLookup*. This procedure is very similar to *Dynalookup* described above. The reader is referred to the source code for the dialog *LOOKTEMP.MAN*.

3.9 Tables of Example Dialogs and Procedures

The following three tables list the names of all dialogs and procedures, their menu choice name, and the features of Maniplus or Manipula they exhibit. Some dialogs and procedures do not have direct menu choices, they are called by other dialogs or procedures. Some menu choices do not call dialogs or procedures. The setup *DIALOGS.MAN*.

Dialog	**Menu choice**	**Features illustrated**
	File	Main menu heading
	e**X**it application	Exits manipulate section and setup
	Lookups	Main menu heading
FixedLook	Browse **F**ixed	- Lookup, fixed ASCII menu file with several fields
SeparatedLook	Browse **S**eparated	- Lookup, separated ASCII menu file with several fields
AnyASCIILook	Browse **A**SCII	- Lookup, file *DIALOGS.MAN* with a field of 120 charac-ters representing each line of text.
SubsetOfFile	S**U**bset of file	-Lookup, fields property displaying a subset of fields. - Several lookup property used
	Controls	Main menu heading
AskUserName	**T**ype your name	-Control, implicit *EDIT* = YES, user enters text string
DisplayUser Name	Display your **N**ame	-Control, *EDIT = NO* allows display of text only. User cannot modify text.

AskPassword	Enter **P**assword	-Control, *EDIT = YES*, and *DISPLAY = NO*, stars cover the password user types.
	Display pass**w**ord	DISPLAY instruction in CASE structure.
EnterID	Enter **I**D number	-Control, *VALIDATEDIRECT = YES* and *DEFAULTBUTTON = NO* for several numeric data cells.
EnterData	**E**nter Some Data	-Control, twelve data cells of various types. -Shows many dialog settings. -Demos how buttons work with controls. -Shows relationship between VALUE and STORE.
DisplayData, calls FillButtonText	Display Some **D**ata	-Control, displays the twelve controls from EnterData. *EDIT = NO* for all controls. -Calls another procedure.
	Texts	Main menu heading
AboutDialog	**A**bout	-Text, shows several text lines making dialog of info.
DisplayName AsTextDialog	Display user **N**ame	-Text, displays user name that was entered elsewhere
	Buttons	Main menu heading
	Arrangement	Sub menu heading

SixButtons	**S**ix buttons	-Button, demos several button display properties such as *CAPTION*, *STATUS*, VALUE, and *HELP*. -Maniplus arranges buttons in dialog.
NineButtons	**N**ine buttons	-Buttons, demos how Maniplus arranges buttons on the right of the dialog if there isn't enough room at the bottom.
SmileyFace	Smiley **F**ace	-Buttons, shows how *POSITION* can arrange buttons into a smiley face.
NineSqueezed Buttons	Nine S**q**ueezed	-Buttons, shows that a small dialog may be too small for all the buttons required.
	Sub dialogs	Sub menu heading
SubDialogWith Buttons, calls ShowSome Buttons	**S**ub buttons	-Buttons, shows *ONPRESS* calling another dialog. -Shows how button-related auxfields hold their value from one dialog to the next. -Shows how button-related auxfields lose their value when its dialog is invoked.
	Mixed dialogs	
AnyASCIILook	**B**rowse any file	-SELECTFILE system dialog to get file name. -Lookup on user determined file.

MenuFile Details	**D**etails screen	-Example of all dialog elements,TEXT, CONTROL, BUTTON, and LOOKUP used in one dialog.
DisplayUser Name	Display user **N**ame	Second call to this dialog from menu.
ShowSome Buttons		-Shows variable text. -Shows how values of auxfield can be passed to another dialog. -A dialog called by another dialog.
FillButtonText		-Shows variable text display using a control

The setup *DIALPROC.MAN*.

Dialog or procedure	**Menu choice**	**Features illustrated**
	File	Main menu choice
AboutDialog	**A**bout this example	-Text dialog.
DosProc, calls ProgName	**R**un DOS command	-Procedure showing *RUN* instruction with various options. -Calls dialog ProgName to get DOS command.
TransmitData	**T**ransmit data	-Procedure held in an INCLUDE file. -2 RUN commands. -PKZIP on files. -OPEN, CLOSE on files to avoid data corruption.

UnzipFiles	**U**nzip files	-RUN commands. -DISPLAY. -RELEASE and OPEN on data files.
	Browse Files	Main menu choice
ASCIILook, calls AnyASCII Look	Browse (Lookup) **D**IALPROC.MAN	-Procedure calls dialog.
BrowseAnyFile	**B**rowse (Lookup) any file	-SELECTFILE. -User selects file to look at.
ButtonStuff, calls ButtonsIn Dialog	B**u**tton stuff with browse (Lookup)	-Calls dialog. -Dialog has 5 buttons executing tasks: Start DEP, type codes, write codes, and search for a form in a data file using a Maniplus search dialog. -*ONPRESS = SEARCH* *-ONPRESS = ProcID* -Text element for user instruction.
	DEP access	Main menu choice
EnterCodesProc calls EnterCodes	**E**nter data without invoking DEP	-WRITE to Blaise data set without invoking DEP. -GET instruction. -Calls dialog. -HIDE = YES. -READY = YES.
LookupData	**V**iew data entered outside of DEP	-Lookup on Blaise data file. -FIELDS sub-element.

GetDepKey, calls EnterID	**G**et ID, non-repeating prompt	-Calls dialog to get ID number. -Computes ID string. -EDIT instruction. -EDIT options, especially to get an old form. -CONFIRM option.
GetDepKey2, calls EnterID	Ge**t** ID, repeating prompt	-Like GetDepKey but with REPEAT -UNTIL structure.
AddnewID, calls EnterID	A**D**d new ID, with CONFIRM	-Calls dialog to get new ID number. -EDIT instruction. -EDIT option to create new form.
GetOldForm	Get **O**ld form, entering ID in DEP	-EDIT instruction. -EDIT option to get existing form. -User enters ID in DEP.
BrowseForm	Get old form, **B**rowsing in DEP	-EDIT instruction. -EDIT option to allow user to browse for a form in DEP, using the DEP browser.
AddNewForm	**A**dd new form, entering ID in DEP	-EDIT option to to create new form, entering ID in DEP.
SelectFormProc calls SelectForm	Se**l**ect form from scroll table	-Calls lookup dialog -User selects form in lookup, invokes DEP directly.

The setup LOOKTEMP.MAN.

Dialog or procedure	**Menu choice**	**Features illustrated**
	Temp file	Main menu choice
DynaLookup, calls FillTempTable, ShowTemp Table.	List the **w**hole file	-Calls procedure. -Calls dialog. -Shows RESET. -Starts off a whole train of events through the two calls.
PartialLookup, calls FillPartialTable and ShowTemp Table.	List a **p**artial file	-Calls procedure. -Calls dialog. -Shows RESET. -Starts off a whole train of events through the two calls.
MakeMenuLine		-Fills one line of the temporary file that makes up the menu file. -Procedure accepts parameters.
CreateMenu, calls MakeMenuLine		-Creates the whole menu file by calling MakeMenuLine repeatedly. -Passes parameters to MakeMenuLine.
FillSortField		-CASE structure. -Makes assignments to the sort field of the temporary file based on choices by user. -Shows how to control order of sort.
FillTempData, calls FillsortField		-INITRECORD -Specifies how to transfer data from the Blaise data file to the temporary file for one line..

FillTempTable		-OPEN, READNEXT, and CLOSE file handling words. -WHILE-ENDWHILE loop. -Calls FillTempData many times to create the temporary file.. -ERASE
SelectSortField		-Control on auxfield. User picks sort field.
SortTempTable, calls SelectSortField, FillSortField		-Calls dialog and procedure. -FOR, DO, RESET, FORMCOUNT, and SETREADKEY, file handling commands -Switches between primary and secondary keys for order of processing.
EditEntry, calls FillTempData		-OPEN, CLOSE, WRITE file commands. -EDIT instruction with options.
ShowTemp Table		-Lookup dialog with fields properties and buttons. -User can invoke DEP and do a sort.
ChooseForm Type		-Control dialog to select subsetting field criterion.
WhichRegion		-Control dialog to get Region number.
FillPartialTable calls ChooseForm Type, WhichRegion, FillTempData,		-Much like FillTempTable above. -IF-ELSEIF. -ERASE.

4. Invoking Programs

A Maniplus application may be called upon to invoke another program including:

- A DOS command such as *DEL*, or *TYPE filename > LPT1*,
- A DOS executable program, such as a text editor, a program for file compression, or a file listing utility,
- The Data Entry Program (DEP),
- Manipula,
- Another Maniplus program.

DOS commands and programs are invoked with the *RUN* function. DEP is invoked with either the *EDIT* method or the *EDIT* function. Manipula and Maniplus are invoked with the *CALL* function.

An example setup *OPROGRAM.MAN*, has several procedures showing these functions and method. To run the setup prepare the following (do not invoke them). For the Maniplus and Manipula setups, use the <F7> key in the control centre. For the Blaise data models use the <F9> key in the control centre.

- Prepare *NCS_EASY.BLA*.
- Prepare *DM_MENU.BLA*.
- Prepare *ASCTOSEP.MAN*.
- Prepare *DYNAMEN2.MAN*.
- Prepare *NCS_DATA.MAN*.
- Prepare *OPROGRAM.MAN*.

Invoke the BAT file *OPROGRAM.BAT* to invoke the Maniplus setup. It is necessary to use the BAT file to invoke the setup in order for one of the procedures to work properly.

There are three major menu headings in the *OPROGRAM.MAN* setup, one each for *RUN*, *EDIT*, and *CALL*. To understand the material in this chapter it is necessary to execute this setup and experiment with it and the procedures it contains.

Replacing BAT files

The functions *RUN* and *CALL* can be used for batch processing (see procedure *CreatMsf* below). This allows Maniplus to replace *BAT* files in *DOS*. This is what a number of experienced Manipula and Maniplus programmers do.

4.1 RUN

The *RUN* function is used to execute *DOS* commands or *DOS* executable programs. The syntax is:

```
RUN (S [, CLEARSCREEN] [, WAIT])
```

where *S* is the *DOS* command or program and the words in brackets are optional. If *CLEARSCREEN* is used, the screen is cleared before execution. If *WAIT* is used, a prompt to press any key will be displayed before returning execution to Maniplus.

The *RUN* function will release as much memory as possible before executing the command. If necessary a new command shell will be loaded. If appropriate, an *EXIT* message will be displayed.

There are six examples of *RUN* in *OPROGRAM.MAN* under the main menu *RUN*. The simplest is in the procedure *DosDir*.

```
PROCEDURE DosDir
  Reslt := RUN('DIR | MORE', CLEARSCREEN, WAIT)
ENDPROCEDURE
```

This procedure is hard-coded to run a directory listing. After all files are listed, a prompt to return to Maniplus is displayed. When a key is pressed, the user is back in the Maniplus application. A similar procedure *DosBDDir* writes a text file of all Blaise data files that is suitable for processing by Maniplus.

The dialog *ProgName* and the procedure *DosProc* allow the user to run a *DOS* command or program. *ProgName* is used to determine the name of the *DOS* command. It also determines if the *CLEARSCREEN* and *WAIT* options should be applied. The result are placed in auxfields *ProgN*, *ClearIt*, and *WaitAfter*.

```
DIALOG ProgName
  SIZE = (55,10)
  CONTROL ProgN
    POSITION = (15,2)
  CONTROL ClearIt
    POSITION = (15,4)
  CONTROL WaitAfter
    POSITION = (35,4)
  BUTTON OneOkayButton
    VALUE = OK
```

The procedure *DosProc* calls the dialog *ProgName*.

```
PROCEDURE DosProc
  ProgName                                    {DIALOG}
  IF OneOkayButton = ok THEN
    IF (ClearIt = yes) and (WaitAfter = Yes) THEN
      Reslt := RUN(ProgN, CLEARSCREEN, WAIT)
    ELSEIF ClearIt = yes THEN
      Reslt:= RUN(ProgN, CLEARSCREEN)
    ELSEIF WaitAfter = yes THEN
      Reslt:= RUN(ProgN, WAIT)
    ELSE
      Reslt:= RUN(ProgN)
    ENDIF
    IF Reslt <> 0 THEN
      DISPLAY('Command "' + ProgN + '" could not be ' +
              'executed', WAIT)
    ELSEIF RUNRESULT <> 0 THEN
     DISPLAY ('Command "' + ProgN + '" gave ERRORLEVEL '
            + STR(RUNRESULT), WAIT)
    ENDIF
  ENDIF
ENDPROCEDURE
```

Depending on the options chosen by the user, the appropriate branch of the *IF - ELSEIF* structure is executed.

If the DOS command or program could not be executed, the value of the auxfield *Reslt* is non-zero. You can make use of this fact and give the user a message with a *DISPLAY* instruction as shown above. Another useful function is *RUNRESULT*. It returns the exit code of the previous command executed under the previous *RUN* function. If the result of *RUNRESULT* is zero, then either the program ran correctly, or a DOS command shell was loaded before the DOS command was executed.

A small Pascal program called *EXITCODE.PAS* has been included with the distribution. If compiled and run from *OPROGRAM.MAN*, it can be used to experiment with non-zero error levels and the messages it causes to be displayed. In the program are comments that explain how to run the experiments.

The procedure *CreateMSF* invokes several *DISPLAY* and *RUN* commands in succession. It calls the Manipula system executable file in order to prepare a Maniplus setup.

```
PROCEDURE CreateMsf
  DISPLAY ('This procedure creates HELLO.MSF ', WAIT)
  Reslt := RUN('DEL HELLO.MSF', CLEARSCREEN)
  Reslt := RUN('MANIPARS HELLO /S /C')
  DISPLAY ('The directory listing of HELLO.MSF.', WAIT)
  Reslt := RUN('DIR HELLO.MSF', CLEARSCREEN, WAIT)
ENDPROCEDURE
```

CreateMsf uses the DOS protected mode program *Manipars.exe* to generate the MSF file. This is not an invocation of a Manipula or Maniplus program, rather the preparation of one. To invoke a previously prepared Manipula or Maniplus setup, use the *CALL* function.

Separate license required

Note that the *CreateMsf* procedure could not be distributed to hundreds of interviewers without paying a hefty license fee as each would need their own license to manipulate meta data. It is included here as an example of how to run a DOS program. It could be appropriate to distribute it to several in-office locations (license permitting) if the locations have a need to prepare Manipula programs. Other procedures in this manual can be distributed to laptops without problems because they do not involve the manipulation of meta data.

Avoiding flicker with DISPLAY

Under certain circumstances , when alternating *DISPLAY* functions with *RUN* functions without the *WAIT* option, the screen may be seen to flicker. In this case, you can use the option *PERMANENT* with the function *DISPLAY* to avoid flicker. The function *CLEARDISPLAY* clears the message window. The

procedures *DemoDisplay1* and *DemoDisplay2* illustrate this point. The first may produce flicker in some circumstances. The second eliminates the problem.

Using RUN when Maniplus is in a DOS window in Windows

It is possible that the Maniplus application will freeze when the *RUN* function is used when Maniplus is invoked in a DOS window under Windows. In this case, you should get your system administrator to look at the problem. It can be fixed by adjusting the Maniplus *PIF* file.

Resident programs

Certain resident programs under DOS, such as the *PRINT* command should not be executed with the *RUN* function due to possible memory conflicts. If it is necessary to run these kinds of commands, you can start them before Maniplus is started, or you can cause Maniplus to quit itself, invoke the program, then come back. The latter topic is covered below.

Recursive *BAT* file

Some DOS executables may not be able to run from Maniplus. In this case, it is possible to cause Maniplus to quit itself, run the DOS executable program, then re-invoke Maniplus. To do this you need a recursive *BAT* file and a way to exit Manipula with a message telling the recursive *BAT* file what to do. Important parts of the example *BAT* file *OPROGRAM.BAT* are listed.

```
@ECHO OFF
:start
CALL ManiPlus OPROGRAM
REM ERRORLEVEL 255 (the number used in the setup)
REM indicates a controlled halt
IF ERRORLEVEL 255 GOTO :again
IF ERRORLEVEL 1 GOTO :error
IF ERRORLEVEL 0 GOTO :end
:error
ECHO a severe error
GOTO end
:again
cls
ECHO Here is where you would call your DOS program.
pause
GOTO start
:end
ECHO finished
@ECHO ON
```

The *BAT* file starts by calling the Maniplus setup *OPROGRAM.MAN*. The line:

```
IF ERRORLEVEL 255 GOTO :again
```

is only invoked if Maniplus is exited with an error level of 255. You can think of *ERRORLEVEL* as a way for a program to send a message to the DOS operating system. An *ERRORLEVEL* of 0 means that a program finished successfully. Other *ERRORLEVEL* numbers mean different things. Maniplus never generates an *ERRORLEVEL* greater than 235. Thus all *ERRORLEVELs* from 236 to 255 can be safely used by the developer. In the Maniplus setup *OPROGRAM.MAN* the procedure *OutAndBack* causes Maniplus to stop and give the error level number 255 to DOS.

```
PROCEDURE OutAndBack
  IF CONFIRM('Are you sure you want to leave Maniplus'
  THEN
    HALT(255)
  ENDIF
ENDPROCEDURE
```

After asking the user to confirm to leave the program, Maniplus stops, giving DOS the error level number *255* through the *HALT(255)* instruction.

Since error level *255* is given to DOS, the *BAT* file branches to the *:again* label. It is at this point that another *BAT* file or executable program can be called. After the other program has been run, the *BAT* file branches to the *:start* label where the Maniplus setup is called again. If the user exits Maniplus in a normal fashion, then the error level number given to DOS is *0*. When this happens the *BAT* file branches to the *:end* label and finishes.

The line of the *BAT* file:

```
IF ERRORLEVEL 1 GOTO :error
```

causes all *ERRORLEVELs* from 1 to 254 to go to *:error*. This is good practice for *BAT* files, to test on the error level and stop if some error is present.

If the *USES* section in the Maniplus setup has a reference to a very large data model, then it can take many seconds to load when the setup is re-invoked. Thus the use of the recursive *BAT* file is discouraged for simpler tasks that can be executed with the *RUN* function.

4.2 EDIT

The *EDIT* method and *EDIT* function are used for invoking the Data Entry Program, to conduct an interview, to edit data, or for data entry.

EDIT method

The *EDIT* method is used when a *USES* section describes an already known data model. In this situation, Maniplus loads the data model when Maniplus itself is loaded into memory. Its syntax:

```
F.EDIT (S)
```

F is a file identifier. The keyword *EDIT* is appended to the file identifier with a full stop between. *S* between parentheses is a string of command line options. These options are given below.

When you use the *EDIT* method, *DEP* is regarded as a sub-process of Maniplus. In other words, it is not necessary to load *DEP* itself. This saves on memory requirements and invokes the data model much more quickly.

In the example setup *OPROGRAM.MAN*, there are seven examples of the use of the *EDIT* method. All of these invoke a data model on the same file known as *HHFile*. They differ in the details about how they accomplish this task.

Procedure	**How it Works**
GetDepKey	- Use dialog *EnterID* to enter ID numbers, invoke the data model on the existing form.
GetDepKey2	- Same as *GetDepKey*, plus - REPEAT the dialog *EnterID* when the user leaves DEP.

AddNewID	- Use dialog *EnterID* to enter new ID numbers, then create a new form.
GetOldForm	- Invoke DEP directly, in GET mode, to bring up an existing form. Use DEP itself to enter the ID numbers.
BrowseForm	- Invoke DEP directly in BROWSE mode. Use the DEP browser to get an existing form.
AddNewForm	- Invoke DEP directly in NEW mode. Let the user enter the new ID in DEP.
SelectFormProc	- Let the user scroll through a lookup table, declared in dialog *SelectForm*, to bring up an existing form.

The method used to invoke *DEP* depends on the needs of the application, and the preferences of the developers and users of it. All seven procedures employ a line of code similar to this:

```
Reslt := HHFile.EDIT(S)
```

where *S* is a text string of options. It is these options that inform *DEP* what it can do.

Testing for an error

In the above example, you can test for the value of *Reslt*. If it is not zero, an error occurred.

Using a Dialog to Enter ID numbers

You can let the user enter ID numbers in a dialog then invoke *DEP* once the ID string is known. This is demonstrated in the procedure *GetDepKey*.

```
PROCEDURE GetDepKey
  Region := EMPTY
  Stratum := EMPTY
  SampleNum := EMPTY
  EnterID                                          {DIALOG}
  HHID := STR(Region) + ';' + STR(Stratum) + ';' +
          STR(SampleNum)
  CASE SomeButtons OF
    ok : IF CONFIRM('Invoke instrument') THEN
           Reslt := HHFile.EDIT('/G /X /K' +
                    HHID)
         ENDIF
  ENDCASE
ENDPROCEDURE
```

The dialog *EnterID* is used to enter the ID numbers into the auxfields *Region*, *Stratum*, and *SampleNum*. The string auxfield *HHID* is computed to hold a concatenation of these values. When the user confirms to invoke the instrument the data model is invoked under the control of some command line options.

```
Reslt := HHFile.EDIT('/G /X /K' + HHID)
```

Command Line Options

Command line options dictate what *DEP* is allowed to do, or pass information such as an ID string to it. In this example, */G* puts *DEP* in *GET* mode. It can only get an existing form, the user cannot enter a new ID in this mode. */X* tells *DEP* to quit after one form is edited. The key option */K* followed without spaces by the ID string, tells *DEP* which form to get. A list of command line options is given below.

EDIT Function

The *EDIT* function is used when it is not known ahead of time which data model is to be used. The *EDIT* function loads the data model at runtime, that is when the user requests it. The data model is not loaded when Maniplus itself is loaded. The *EDIT* function syntax is:

```
EDIT (S)
```

where *S* is a string representing command line options. Two procedures in the example setup *OPROGRAM.MAN* demonstrate

the use of the *EDIT* function. They are *SelectDep* and *LoadDep*. Both make use of *SELECTFILE* to choose a data model to invoke. *SelectDep* loads and unloads the chosen data model each time the procedure is called. With the *LOADDATAMODEL* instruction, *LoadDep* loads the data model once and holds it in memory until the user causes it to be released. The code for *LoadDep*:

```
PROCEDURE LoadDEP
  DEPModel := SELECTFILE('Choose a data model','',
                         '*.~MI', FIXED)
  IF DEPModel <> '' THEN
    Reslt := LOADDATAMODEL(DEPModel)
    REPEAT
    Reslt := EDIT(DEPModel + ' /X')
    UNTIL CONFIRM('Leave and unload the data model?')
    RELEASEDATAMODEL(DEPModel)
  ENDIF
ENDPROCEDURE
```

The user will leave *DEP* every time a form is exited, but the data model remains in memory. Only if the user does not confirm to continue will the data model be unloaded with the instruction *RELEASEDATAMODEL* and the procedure stopped. Like for the *EDIT* method, the auxfield *Reslt* can be used to test the success of the function.

/F option

The */F* option is not allowed for the *EDIT* method. If it is present it is ignored. Maniplus knows the data file through the meta data in the *USES* section and the data file name in the file section. The */F* option is allowed for the *EDIT* function.

Caution

It is possible to use a *RUN* command to invoke a Blaise data model, but this is not the correct way to do it. The *RUN* function would cause Maniplus to load *DEP*, then load the data model. This puts even more demand on memory and is even slower for the user to invoke the Data Entry Program. Use the *EDIT* method whenever possible, and if necessary, use the *EDIT* function.

Legacy Blaise 2.5 Instruments

If you have a legacy Blaise 2.5 instrument and you want to invoke it from Maniplus, then use the *RUN* function, or exit Maniplus with a recursive *BAT* file (explained above) and execute it from *DOS*. In fact, for a legacy Blaise instrument, this may be a good option due to its memory requirements. Some of the legacy instruments prepared in *REAL* mode could not run in the *DOS* shell provided by Maniplus.

Options for Invoking DEP

Following is a table of options that can be used when calling *DEP*, either from the command line of *DOS*, or from Maniplus when an *EDIT* method or function is used.

Option	**Explanation**
/A<file name>	Use DEP modes file instead of default (<meta file name>.~DM)
/B	Browse (already-existing) forms
/C<file name>	Read configuration file, DEP.DC is default
/D	Disable CATI (if necessary)
/F<file name>	Set main data file, <meta file name>.~BD is default
/G	Get form mode. Reads form in combination with /K.
/J	Start form with given JoinID (only in combination with /K)
/K<key value>	Get the first form with specified PRIMARY key or JoinID
/L<ID number>	Set interview language, first language (1) is default.
/M<file name>	Read menu file, DEPMENU.~MU is default.
/N	Create new form with given key (only in combination with /K).
/P<number>	Set start-up page layout (1 is default)
/T<number>	Set DEP behaviour toggles (1 is default)
/X	Quit DEP after editing first form.

Inspect the example setup *OPROGRAM.MAN* for several examples of how these are used. The Developer's Guide has an example of

invoking the data entry program from DOS with command line parameters, in Chapter 6.

4.3 CALL

To invoke a Manipula setup for a batch job, or to invoke another Maniplus setup from a main Maniplus setup, use the *CALL* function. Its syntax:

```
CALL (S)
```

S is a string of text representing command line options for Maniplus or Manipula. Four procedures demonstrate the *CALL* function. They are *ASCIIToSep*, *GenerateData*, *CallHello*, and *CallDynaMenu*. The first two call Manipula programs, the last two call Maniplus programs. *GenerateData* uses a dialog to obtain information from the user.

Reasons to execute Manipula programs include:

- Reading out completed forms on a *CAPI* laptop computer for transmission.
- Reading in new forms to a data set, for example new assignments sent from the home office.
- Generating data for practice interviews.

Reasons to execute Maniplus setups include:

- A parent setup has several child setups, one for each survey, in a multi-survey *CAPI* laptop management system. By putting each survey in its own setup, only the appropriate data models are loaded as they are needed. This is also a good way to handle general Blaise practice instruments.
- A parent Maniplus setup has several child setups, one for each lookup dialog in a top-down (statistical) editing scheme. This holds down the number of file sections and procedures in any one setup.

The procedure *ASCIIToSep* calls a Manipula program that creates an separated ASCII file from a fixed ASCII file.

```
PROCEDURE ASCIIToSep
  IF FILEEXISTS('ASCTOSEP.MSF') THEN
    Reslt := CALL('ASCTOSEP')
  ELSE
    DISPLAY('ASCTOSEP.MSF file must be prepared.', WAIT)
  ENDIF
ENDPROCEDURE
```

The procedure checks that the Manipula setup *ASCTOSEP.MAN* has been prepared. If so, the file *ASCTOSEP.MSF* exists and it is invoked with the *CALL* function. In this example, the developer has taken care of providing a message in case the file had not been prepared.

The procedure *CallDynaMenu* calls the Maniplus prepared setup *DynaMen2.Msf*.

```
PROCEDURE CallDynaMenu
  Reslt := CALL('Dynamen2')
ENDPROCEDURE
```

Here, the developer relies on the system to generate its own message if the setup file is not prepared.

The result of a CALL

The result of a *CALL* function is the exit code from the Manipula/Maniplus setup executed. For example, a *HALT*(245) will result in *Reslt* having the value 245. This gives the possibility to communicate something back to the parent setup.

Loading Data Models

If the called Manipula and Maniplus setups have *USES* sections with Blaise data models mentioned, then those data models are loaded when the setup is called. If the called setup uses data models already loaded by the parent, the data models are not newly loaded. In this case, the data model already loaded is used.

Recursion not allowed

Recursion of *CALL* functions is not allowed. For example, A calls B, B calls C, and C calls A, is not allowed.

Options for Calling Manipula or Maniplus

Following is a table of options that can be used when calling Manipula or Maniplus, either from the command line of DOS, or from Maniplus when an *CALL* function is used.

Option	**Explanation**
/A	Wait for key at end.
/C	Check setup, do not run.
/D<file name>	Set DAY file name.
/E	Execute MSF file.
/F<dir name>	Set read-from directory.
/H<dir name [;dirname]>	Set meta search path(s).
/L<language>	Set system language.
/M<filename>	Set parameter file name.
/O<filename [,filename]>	Set output file name(s).
/P<parameter [;parameter]>	Set parameters.
/Q	Run in quiet mode.
/R	Set message file name.
/S	Save MSF file.
/T	Set write-to directory name.
/W<dirname>	Set execute work directory.
>Fname	Write event log to the file Fname. Delete Fname if it exits before writing it again.
>>Fname	Write event log to the file Fname, appending new information to old.

The last two entries are not command line options per se. You should list them after the true command line options. This way you can generate a file of events. The information contained here complements the information in the day file and the message file.

Inspect the example setup *OPROGRAM.MAN* for several examples of how these are used. The Developer's Guide has an example of invoking Manipula from DOS with command line parameters, in Chapter 7.

4.4 Dynamic Link Libraries

Dynamic Link Libraries (DLLs) are covered in the Developer's Guide and in text files provided in the Blaise system distribution. They allow you to do things that Maniplus or Manipula cannot do. An example is to scramble a password in a file so that an intruder cannot look at it if a laptop computer is stolen. Two example setups are provided. They are *PASSWORD.MAN* and *CHCKCOPY.MAN*.

It is up to the user to secure appropriate technical expertise when using DLLs. They are not for the novice. You must test them thoroughly before putting them into production. Make sure you read all of the documentation provided by the Blaise developers before proceeding.

5. Handling Files

Usually Maniplus takes care of the file handling for Blaise data sets and other files. There are, however, situations where it is best to take care of file access and control yourself. The first situation is when *OPEN = NO* is a setting in a file section. For example, from *LOOKTEMP.MAN*:

```
UPDATEFILE HHFile1 : NCS_EASY ('NCS_DATA', BLAISE3)
  SETTINGS
    OPEN = NO
    ACCESS = SHARED

INPUTFILE HHFile2 : NCS_EASY ('NCS_DATA', BLAISE3)
  SETTINGS
    OPEN = NO
  FILTER
    Ident
    Manage
    Address
```

The file is not opened until later when the application needs it. In this case it was done this way because two different file names refer to the same file. Another reason *OPEN = NO* is used is in situations where the file name is not specified in the file section.

It is best to personally take control of file handling when an external program is going to do something with the files. For example, you may want to compress the files before transmission. You must make sure that the file is closed and released before you try to compress it. Otherwise the compression software may get very confused as it tries to compress something in use by another system.

5.1 Table of File Handling Commands

Following is a table of file handling commands that have been used in various examples. Many more commands are available. A complete list of 41 file commands for Manipula and Maniplus is given on page 435 of the Reference manual. Note that the command *RELEASE* is not documented in the Reference Manual but is included in the on-line help.

File Key Word	Example Setups	Explanation
CLOSE	DIALPROC, LOOKTEMP	Closes a data file but does not release it.
DELETE	DYNAMENU, DYNAMEN2	Deletes a form from a file, at the current pointer position. This is usually used with RESEARCH and GET or some other key word that locates and reads the correct form.
ERASE	LOOKTEMP	Erases a whole data file. You do not want to use this on a survey data set, but it is useful when a temporary file or updatefile is used as a menu file.
FILEEXISTS	LOOKTEMP	Checks if a file exists on disk. If you need to check that a temporary file exists, use FORMCOUNT and check for a value greater than zero.
GET	DIALOGS, DIALPROC	Searches and reads a record. Puts the file pointer on the record.
INITRECORD	LOOKTEMP	Initialises output. Clears a record in a file, usually so when it is filled again, only new data are present.
OPEN	DIALOGS, DIALPROC, LOOKTEMP	Opens a file that had been closed or never opened. Useful when switching between file, or when two file sections refer to the same file.

RESET	LOOKTEMP	Places the file pointer to the first record in the file.
RELEASE	DIALPROC	Closes and releases a file. This allows another application to process the file, for example, a file compression utility.
RESULTOK	DIALPROC, LOOKTEMP	Used to test if the previous file access instruction was valid. For example, if you use GET to search and read a form, you can use RESULTOK to make sure that the GET instruction was useful. If it was not, you might display a message to the user.
WRITE	DIALPROC, LOOKTEMP, DIALDEMO, others	Writes a record to a file. If you try to enter data into a Blaise data file from Maniplus, you use the WRITE to make sure that the data are written to disk.

ACCESS Setting in Manipula and Maniplus

The file setting *ACCESS = SHARED* is meant to allow Manipula and DEP to share the same data file simultaneously. *ACCESS = EXCLUSIVE* (the default) means that Manipula or Maniplus has exclusive control of the data file. For technical reasons, *ACCESS = SHARED* is used in *LOOKTEMP.MAN* in certain cases in order to give more protection to the data file in case of laptop failure.

With *ACCESS = SHARED* file handling is markedly slower than with *ACCESS = EXCLUSIVE* because the file is opened and closed after each form is read. It is for this reason that file based operations (e. g., reading the data set) in *LOOKTEMP.MAN* are handled with the default setting *ACCESS = EXCLUSIVE* and the form based operations (e.g., invoking *DEP* on one form) are handled with *ACCESS = SHARED*.

DOS SHARE

It is recommended that the *DOS SHARE* utility always be loaded in a stand-alone environment. *DOS* uses the code loaded by *SHARE* to validate all read and write requests from programs. This protects you from undesired and uncontrolled multiple use of a file. This may happen, for example, if one procedure does

not close a file before another procedure to compress it is invoked. In a network environment, *SHARE* should not be loaded. Its function is already provided for in a network and its use may slow down network performance.

6. HELP Facility

You can easily invoke a hyper-text link help facility for the Maniplus application user. With this you can:

- Give detailed information on each menu selection, dialog, procedure, and button.
- Give the user help in general survey procedures.
- Give the user explanations of the justification for the survey that may help her with persuading a respondent to participate.

The appropriate help topic is displayed in a pop-up window when the <F1> key is pressed. Within the help window, you can give the user links to other help topics. When the cursor is put on the highlighted topic and the <Enter> key is pressed, the help window for the alternate topic will appear.

The help facility is available throughout Maniplus. The implementation of the help facility is explained first with reference to menus. This is done with the example setup *ASCMENU.MAN* which is one of the menu examples (Chapter 2). It is very easy to extend the help facility to other Maniplus elements. This is demonstrated with the example setup *OPROGRAM.MAN* which is the main example for invoking other programs (Chapter 4). It shows how to implement the help facility in dialog elements and in the manipulate section. It also demonstrates how to implement help where a parent Maniplus setup calls a child Maniplus setup.

6.1 Building the HELP System for Menus

To implement the help facility for menus, follow a simple recipe:

- Enable the help facility in the Maniplus setup.
- Type help numbers in the menu file for each menu choice where you want help to be available.
- Type an ASCII text help file following a particular format.
- Convert the ASCII file to a binary file with extension *HLP* using the utility *MANIPHLP.EXE*.

Enable the help facility

To enable the help facility in Maniplus, use the help setting at the top of the setup. For example, from *ASCMENU.MAN:*

```
SETTINGS
  HELP = YES
  HELPFILE = 'ASCMENU.HLP'
```

With the *HELPFILE* setting, you can name any file to be the help file, as long as it has been formatted correctly into a binary help file with *MANIPHLP* (see below). If the help file has the same forename as the setup, *HELPFILE* is not necessary.

Type help numbers in the menu file

When creating the menu file, you type help numbers corresponding to the help number that will be entered in the ASCII text help file. One easy numbering scheme is the convention employed in the *ASCMENU.ASC* example. The following entries are made for one record in the menu file.

```
1.2      {The MenuID Number}
12       {The help menu number}
```

or for a high level choice:

```
2      {The MenuID Number}
20     {The help menu number}
```

If the help number is equal to the *MenuID Number* (but without the dots) then you have an easy, consistent numbering scheme.

Type an ASCII text help file

Type the help instructions corresponding to each menu entry with a help number in an ASCII text help file. An example of an ASCII text help file is *ASCMENU.TXT*, one entry of which is shown here.

```
.topic Inst = 20
 Instrument menus
 _______________
These are menu and submenu items that are used to select
an instrument for actual interviewing and practice
interviewing. You can select the case by using the point
and shoot method or by entering the ID number. You can
also add new forms if necessary.

See also:
 {Survey instrument:ActInst},
 {Practice:Practice}
```

The formal syntax of a topic entry in the help file is as follows:

```
.TOPIC Ident [ = Number ] [ , Ident [= Number] [ ... ] ]
 Topictext
```

Each topic's entry is started by the key word *.topic* (the dot must appear before *topic*). An identifier appears next followed by a help number. If the help number is not typed, then the help facility will assign a number which is one more than the previous number. For the entry above:

```
.topic Inst = 20
```

the identifier is *Inst* and the help number is *20* (which also appears in the menu file).

The topic text can be almost anything you want it to be. For this example, topic text is anything below the first line. To make it more readable for the user, a heading is given followed on a second line by an underscore.

```
Instrument menus
_______________
```

The heading and the underscore are not required. They are stylistic conventions that may make it easier for users to understand the topic quickly. Note that both header lines begin with a space to avoid invoking an automatic wrapping feature.

After the headings a few lines are typed explaining the entry. Then there is mention of related entries that the user might want to go to. These are implemented by means of hyper-text-links.

```
See also:
 {Survey instrument:ActInst},
 {Practice:Practice}
```

Once again, in order to get the lines to appear in the help window on their own lines, the lines were started with a space.

The implementation of this help window is shown in figure 6.1.1.

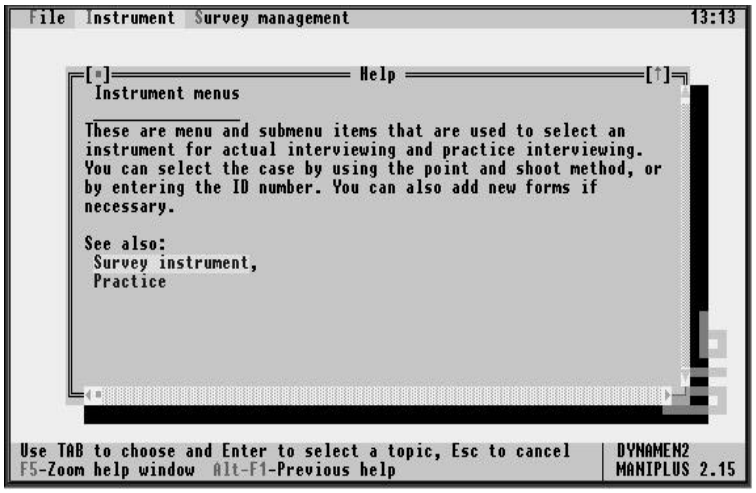


Figure 6.1.1. An example of a help window

The hyper-text-links can be placed anywhere in the text of the help entry, not just at the end as shown here. The syntax of declaring a link is:

```
{ Text:Ident }
```

The characters **{** and **}** are part of the syntax. The *Text* will appear highlighted in the help window. *Ident* is the identifier of the help topic that is activated when *Text* is selected. The user selects a text by tabbing, with arrow keys, or with a mouse.

An example of a topic text in which a reference is made:

```
{Survey instrument:ActInst},
```

The text *Survey instrument* will be highlighted. If the user places the cursor on that text string and presses <Enter> the help window associated with the identifier *ActInst* will be displayed.

Making a binary help file

Once the ASCII text help file is created it is necessary to convert it to a binary file with the *HLP* extension. Assuming you've used the *TXT* extension for the ASCII file, and that the Blaise system directory is in the path, use the following command.

```
MANIPHLP TextFileName
```

for example:

```
MANIPHLP ASCMENU
```

The extension *TXT* for the file *ASCMENU.TXT* is assumed. The file *ASCMENU.HLP* is generated and is used directly by Maniplus to implement the help facility. You can run the program *MANIPHLP* from the control centre, using the *Tools*, *Own program* menu choices.

Help with dynamic menus

The setup *DYNAMEN2.MAN* is an implementation of dynamic menus. It has two menus, one for supervisors, and one for interviewers. When you implement the help facility numbering scheme with dynamic menus, remember to implement it in both menus consistently. See the procedures *SuperMenu* and *InterMenu* for examples.

Another feature of the setup *DYNAMEN2.MAN* is that the *About* box dialog can be alternately deleted and added, or enabled and disabled. The procedures that implement these dynamic features must also implement the help topic ID consistently. See the procedures *A_Proc1*, *A_Proc2*, and *A_Proc3* for examples.

6.1.1 Default System HELP Numbers

You can use any number from 1 to 10,000 for help numbers, except for reserved numbers from 1000 to 1024. The help facility gives you default entries with these numbers. They are used for common features found throughout Maniplus. These default entries are found in the file *MANIPHLP.HCT*. Two of them are:

```
.topic General = 1000
Press F10 or one of the short cut keys to activate the
menu.

.topic MessageOk = 1001
Push the OK button to continue the current action.
```

You can copy these entries into the ASCII text help file that you have created. You can change the topic text of the above entries but you cannot change the identifiers or numbers. You can find these reserve numbered topics at the end of the help files *ASCMENU.TXT*, *OPROGRAM.TXT* and *DYNAMEN2.TXT*.

6.2 HELP Instruction

HELP can be used as an instruction in the manipulate section or in procedures. This is a good way to display a help index, to give general information about the procedure, or to give general information about the survey. An example of a *HELP* instruction is given in the following *CASE* statement in the example setup *OPROGRAM.MAN*:

```
CASE program OF
  'about'       :     AboutDialog    {DIALOG}
  'reasons'     :     Help(14)       {Topic 14 of help.}

  'dos'         :     DosProc        {PROCEDURE}
 . . .
ENDCASE
```

Help(14) leads to a part of the help facility which lists some reasons why respondents should co-operate with the survey. It is associated with *MenuID* 14. The user can invoke it under main menu choice *File* and sub-menu choice *Reasons for survey*. When the cursor is on that menu choice, the help facility can be invoked either by pressing <F1> or by pressing <Enter>.

Help in Procedures

The help facility can be invoked automatically in a procedure with the HELP instruction. The procedure *BrowseForm* is found in *OPROGRAM.MAN*.

```
PROCEDURE BrowseForm
  HELP(351)
  Reslt:= HHFile.EDIT(' /FNCS_DATA /X /G /B')
ENDPROCEDURE
```

The help topic *351* gives the user an explanation of how to use the browse facility of the Data Entry Program. The help window associated with topic *351* is invoked automatically every time the procedure is invoked. When the user leaves the help window by

pressing the <Esc> key, the *DEP* browser appears. This procedure is invoked by the user from the main menu choice *Edit for DEP* and sub-menu choice *Get old form, browsing in DEP*.

Help in Dialogs

It is possible to implement the help facility in dialogs, and in controls, lookups, and buttons in dialogs. An overall help topic can be declared for the dialog and more specific help topics can be declared for the dialog elements and sub-elements. The dialog *EnterID* is found in the setup *OPROGRAM.MAN*. It appears in the first three sub-menu choices under main menu choice *Edit for DEP*.

```
DIALOG EnterID "Enter an ID Number"
  HELP = 300
  SIZE     = (65, 12)
  DEFAULTBUTTON = NO
  VALIDATEDIRECT = YES
  CONTROL Region
    HELP       = 301
    LABEL      = ('Region:  ', 2, 2)
    POSITION   = (25, 2)
    EDIT       = YES
    STATUS     = 'Enter region of the form, 1 to 97.'
  CONTROL Stratum
    HELP       = 302
    LABEL      = ('Stratum:  ', 2, 3)
    POSITION   = (25, 3)
    EDIT       = YES
    STATUS     = 'Enter the stratum number, 1 to 997'
   CONTROL SampleNum
     HELP       = 303
     LABEL      = ('Sample number:  ', 2, 4)
     POSITION   = (25, 4)
     EDIT       = YES
     STATUS     = 'Enter sample number, 1000 to 9000'
   BUTTON SomeButtons
     CAPTION = '  ~O~kay    '
     VALUE   = ok
```

The help topic *300* is available to any element or sub-element which does not have its own help topic. For example, the window associated with help topic *300* appears if the user presses <F1> when the focus of the dialog is on the *Okay* button. Each of the controls has its own help topic which is invoked when the user is on one of them and presses <F1>.

ProgName is an example of a dialog where all the elements have their own help topic and there is no overall dialog help topic. This dialog is invoked in the Maniplus setup *OPROGRAM.MAN* through main menu choice *Run DOS programs*, and sub-menu choice *Type DOS programs*.

Help topic numbering scheme for dialogs, procedures, and the manipulate section

To ease maintenance, it helps to have an easy-to-read numbering scheme for help topics in the manipulate section, procedures, and dialogs. The scheme used in the setup *OPROGRAM.MAN* is an extension of that used for menus described above. For example, the *MenuID* of the main menu choice *Edit for DEP* is *3*. Its help topic ID is *30*. The *EnterID* dialog which is used in three of its sub-menu choices has overall help topic ID *300* and its sub-elements have topic numbers *301*, *302*, and *303*. It does not matter which numbering scheme you use, it vastly eases development and maintenance if you use one.

6.3 Help in Parent and Child Maniplus Setups

Some implementations of Maniplus have a parent Maniplus setup calling a child Maniplus setup. For example, a multi-survey CAPI laptop management system may have a parent setup where the interviewer can choose a survey to work on and child setups which are specialised for the chosen survey. The help facility can be implemented for both the parent and child setups.

Sharing a help file

The parent and child setups can share the same help file. This implies that they share the same numbering scheme for common topics or menu choices. Where a large parent setup calls a small child setup, it may be easiest to implement the help for the child by using the parent help file. However, where the child setup is large, it is better to have separate help files. If the child setup has a separate help file (hence separate numbering scheme) it is easier to implement the child setup under a different parent setup, or by itself, than if the child setup shares a help file.

Separate parent and child help files

The example setup *OPROGRAM.MAN* is a parent setup to *DYNAMEN2.MAN*. Since the latter setup is large, separate help files are used for each. The developer-created ASCII help files are *OPROGRAM.TXT* and *DYNAMEN2.TXT* respectively. The generated binary files are *OPROGRAM.HLP* and *DYNAMEN2.HLP*. Two other ASCII files are produced in the conversion process. They are *OPROGRAM.HCL* and *DYNAMEN2.HCL*. They list help topics and their numbers and are for reference only. They are not needed by the application.

7. CAPI Laptop Management

Maniplus can be used to powerfully and flexibly manage a *CAPI* survey on laptop computers. A laptop management system should enable an interviewer to:

- quickly invoke the instrument and conduct an interview on the proper form,
- select groups of forms on provided criteria, display them in a lookup table, and allow the forms to be sorted in various ways according to interviewer preferences,
- give the interviewer a summary of the forms' statuses,
- allow the interviewer to generate and inspect reports,
- update status codes for each form,
- transmit and receive forms to and from the home office,
- make appointments and see summaries of them,
- provide general survey information, for example, reasons for conducting the survey that the interviewer can use to persuade a respondent to participate,
- provide information to the interviewer about getting around difficulties to help reduce the need to call to the home office for help,
- use the help facility in order to use the laptop management system more effectively,
- use function keys so that the interviewer does not have to navigate the menus for common tasks,
- allow access to utilities that can enable the interviewer, working in conjunction with the home office, to determine the cause of computer problems,
- send time and expense reports to the home office, and
- practice the instrument and practice using the Blaise system.

There are several features common in laptop systems that are beyond the scope of this example. For example:

- The laptop management system should be secure from intrusion so that if someone else obtains the laptop they cannot get into confidential data. This can be accomplished with a password that must be typed before entering the laptop management system.
- A necessity for some surveys is the ability to transfer and receive forms to and from other interviewers.
- Status codes for the instrument in this survey are not very extensive. Most organisations have their own status code schemes for reporting purposes and general survey management. Some surveys implement the idea of temporary status codes and final status codes. For example, a refusing respondent may be given a temporary refusal status code so that someone else may try this respondent later. Only after a second refusal, or at the end of the survey period, would this form be given a permanent status of refusal.
- Some organisations may want to put sampling frame management into the system, for example, to add a housing unit when a listed one has been divided.
- The management system can be programmed to implement complex subsampling rules.
- A log file can be kept so that home office personnel can inspect the sequence of events when trying to solve a problem. This is documented in Chapter 8 of the Developer's Guide.

Other features can be built into a laptop management system by different organisations. The following example can manage a simple *CAPI* survey, and with modification and extension, manage a complex one.

7.1 Example set-up

An example set-up that includes the above functionality is called *CAPILAP.MAN*. It is found in the directory *\BLAISE31\DOC\MP_CHAP6*. All source code files and a setup to generate an example data set are included in the Blaise distribution. The *README.TXT* file explains how to set up the example. In the set-up *CAPILAP.MAN* some information is given how key features were constructed.

Sub-directories

In order to keep files and data sets organised it is helpful to introduce a subdirectory structure. Due to the way that Maniplus and the Data Entry Program (DEP) work, it is necessary to state fully the directory paths of all instruments and Manipula/Maniplus set-ups that are called from *CAPILAP.MAN*. If the directory structure of your computer does not match that mentioned above, it is necessary to go into the code and adjust the directory paths. Remarks in the example set-up explain where these changes must be made. The subdirectory structure of the *CAPILAP.MAN* example is:

```
SETTINGS     {interviewer settings file}
CAITRAIN     {general Blaise training instruments}
TRANSMIT     {programs to extract forms for
              transmission and a place for data sets
              waiting to be transmitted}
RECEIVE      {a place for files when they are first
              received from the home office}
EXTERN              {miscellaneous files such as
             message files  or remarks files}
INST         {survey instrument}
DATA         {survey data}
PRACDATA     {Practice data}
TIMECOST     {Time and cost data model and data}
```

Single-survey system

CAPILAP.MAN is an example of a single-survey laptop management system. Some organisations construct multi-survey laptop management systems. A brief discussion of how a multi-survey system may be built in Maniplus is given at the end of this chapter.

Doorstep management

Organisations differ on whether they use the laptop system solely as an instrument and data management system or additionally to manage the screening process, appointments, sampling frame information, and more. For the former, the interviewer still needs many paper materials. For the latter, the materials are all on the laptop, but would require the interviewer to have the laptop turned on more. For example, an interviewer may arrive at a door and be able to make an appointment, or find that a housing unit has been divided into two or more units. If the laptop is not used to manage these tasks, then they have to be managed on paper. If all information are recorded electronically, it is easier for the home office to keep track of all kinds of survey progress and of updates to the sampling frame. However, the configuration of the hardware plays an important role in making decisions. The 2, 3, or 4 kilogram 'clam-shell' laptop, with a battery life of 2 hours, is not the best piece of equipment for doorstep work. It may be too heavy and awkward for the interviewer to use while standing, and the interviewer may fear that the battery will expire before the computer can be plugged in.

A workable compromise may be to record some information on paper and have the interviewer transfer it to the laptop management system at home on a daily basis before transmitting data to the office.

Location of management data

Survey management data such as status codes and appointments can be incorporated into the survey instrument or maintained as separate Blaise data models and files. This example maintains such data as part of the data model. This is perhaps the easiest way to program such a system but it may suffer from some inflexibility. One reason this example keeps the administration data within the instrument is to maintain *CATI/CAPI* compatibility. By keeping the management information in the instrument, the *CATI* call scheduler can use it directly when data are transferred into the office Thus multi-mode surveys can make maximum use of in-office *CATI* and in-field *CAPI* computers and make them work together. See the sections below on appointments and multi-survey management systems for reasons to use external files to hold administrative data.

7.2 Advantages of using Maniplus to write a laptop management system

There are several advantages of using Maniplus as a laptop management system. These include:

- Maniplus has knowledge of the meta data of the instruments. This saves you from writing data descriptions to transfer management data to the laptop management system.
- Maniplus loads all instruments into memory when it is started. For some large applications, the loading of a data model can take a long time. By loading data models when the Maniplus system is invoked, the interviewer can start an interview immediately instead of waiting for initialisation in front of the respondent.
- Maniplus can read and write data directly to Blaise data sets.
- Maniplus is an interactive extension of Manipula, thus it has all of the power of Manipula.
- Maniplus has all of the functionality needed for laptop management, for example interactive lookup tables.
- The Data Entry Program and Manipula are part of Maniplus. Thus it is not necessary to load these modules separately when invoking them.

In addition, Maniplus was written by people who have had prior experience in developing *CAPI* management systems. This expertise was put to good use when designing Maniplus.

7.3 Structure of the example laptop management system

The structure of a laptop management system can inspire passionate debate and be subject to much revision as different groups of people are exposed to it. It may be a good idea to get interviewer feedback on it early in the process. The structure can be manifested either through the menu system, through dialogs, or both. This example uses the menu system as the primary organising mechanism and dialogs as a subsidiary one. The structure of the laptop management system for this example is as follows. The lines below denote menu choices except where dialogs are used as noted.

```
File
  About
  Reasons
  Change settings
  Time, Travel, Other Costs
  Exit application
Instrument
  Interview
    Sort and Select ID
      Dialog to select groups of forms
        Lookup of selected forms that:
          - can be sorted,
          - an interview can be started from,
          - a transmission status can be set
    Enter known ID
    New form
  Appointments
      Dialog to select groups of forms
        Lookup of selected forms that:
          - can be sorted,
          - an interview can be started from,
          - a transmission status can be set Practice
    Practice General Blaise
      Blaise CAPI Practice 1
      Blaise CAPI Practice 2
      Blaise CAPI Practice 3
    Refresh practice forms
    Practice survey instrument
Special information
  Special messages
  Remarks
    Read Out Remarks
    View Remarks
      Lookup dialog where the interviewer:
        - can read remarks
        - invoke the instrument on a form with remarks
  Frequently Asked Questions
Communications
  E-mail messages
  Readout Forms
  Dial home
  Hot transmit
Utilities
  Monitor Data Base
  DOS program
```

The reason that this example relies so much on menus is that they are easy to build and because they allow the interviewer to see easily how the laptop management system is structured.

TEMPORARYFILE for the menu system

The menu system in *CAPILAP.*MAN is implemented through a *TEMPORARYFILE* file section and internally through the procedures *CreateMenu* and *MakeMenuLine*. This is done so that no one can tamper with the menu system which could happen if the menu items were held in an external file. It is also easier to maintain since all information is in one set-up.

7.3.1 File

The top-level menu selection *File* contains sub-menu choices *About*, *Reasons*, *Change settings*, *Time*, *Travel*, *Other Costs*, and *eXit application*. These are miscellaneous tasks that are general in nature and do not fit in well with other menu items.

About

The *About* dialog gives general information about the example. It is not necessary from an interviewer standpoint for most applications. However, for multi-survey management systems, it is an easy way to inform the interviewer which survey is currently invoked. The *About* dialog is invoked by the procedure *AboutDialog*.

Reasons

If the laptop management system is used as a doorstep system, that is, the computer is on when the interviewer approaches the household, the *Reasons* feature may aid the interviewer in explaining to a respondent why he or she should participate. This feature is implemented through the help facility where the hyper-text links allow the interviewer jump easily from one topic to another to answer the respondent's specific questions which may come up in any order. The reasons are held in the help file which are accessed with the following command in the Manipulate paragraph:

```
'reasons'       :     HELP(13)
```

See Chapter 6 for information about the help facility.

Change settings

This option invokes a subsidiary instrument where information can be recorded about the interviewer on each laptop computer. In this example, the settings include entries for interviewer number, kind of job (interviewer or supervisor), languages they can interview in, and whether a proxy is allowed for respondents in this survey. The main survey instrument can be set up to read interviewer information from this file which in turn can be used to condition the routes, checks, or allowed languages in the interview.

It is possible to incorporate this information with DOS environment variables that are set on each interviewer's machine. An advantage of using an instrument to record this information is that it allows interviewers to change settings when laptops are traded around. A disadvantage may be that interviewers can change this information too easily.

This setting is called by the procedure *ChangeSettings* in the set-up *CAPILAP.MAN*. It calls the instrument *IntSettings* which is held in the subdirectory *SETTINGS*. In the main instrument, the interviewer is asked for their interviewer identification. If it matches an entry in the settings file the information about languages and proxy is brought into the main data set. Otherwise the interviewer has to enter that information.

Time, Travel, Other Costs

An important advantage of *CAPI* is the ability to quickly transmit data back to the home office. Among the data that can be transmitted are ongoing interviewer costs. This allows the home office to keep track of survey expenses in a timely manner. Many organisations require that such information be recorded on paper for legal reasons. In this case, this cost information can still be recorded electronically and regarded as preliminary until the paper materials come in. This is valuable because the organisation can keep track of costs on a day-to-day basis and not wait every two weeks for a time report to be filed which then has to be key entered. This feature is invoked with procedure

TimeCost. The data model it accesses is *TIMECOST* held in subdirectory *TIMECOST*. This data set is arranged by pay period. Within each pay period, the interviewer can choose from a list of surveys for which to record costs.

eXit application

This is a standard Maniplus feature that allows the interviewer to leave the menu system. Without it, the interviewer would have to turn off the computer in order to leave the system.

7.3.2 Instrument

The *Instrument* main menu selection has sub-menus *Interview*, *Appointments*, and *Practice*. The choices here are the main ways that the interviewer can access a form and either conduct an interview, review a chosen form, or practice with the Blaise system or with the survey instrument on practice data.

Interview

The *Interview* choice has three sub-menu choices, *Sort and Select ID*, *Enter known ID*, and *New Form*.

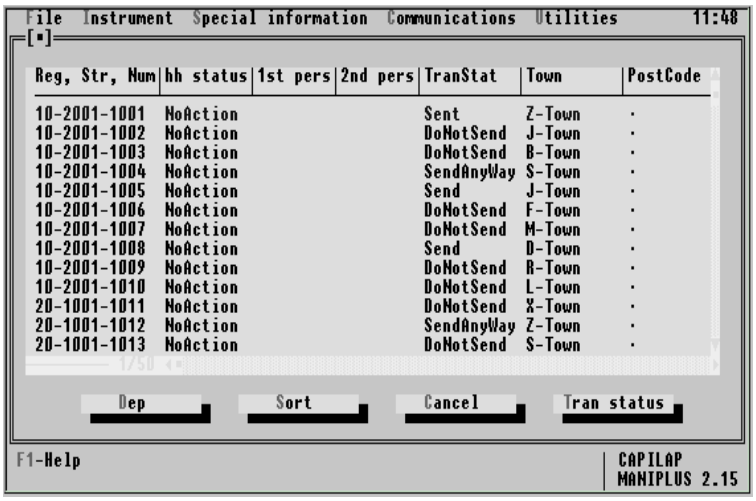
Sort and Select ID

The dialogs invoked by this menu choice allow the interviewer to select and sort groups of forms according to any of the criteria provided by the programmer. When *Sort and Select ID* is invoked the following selection dialog appears.

Figure 7.3.1: Sort and select ID dialog

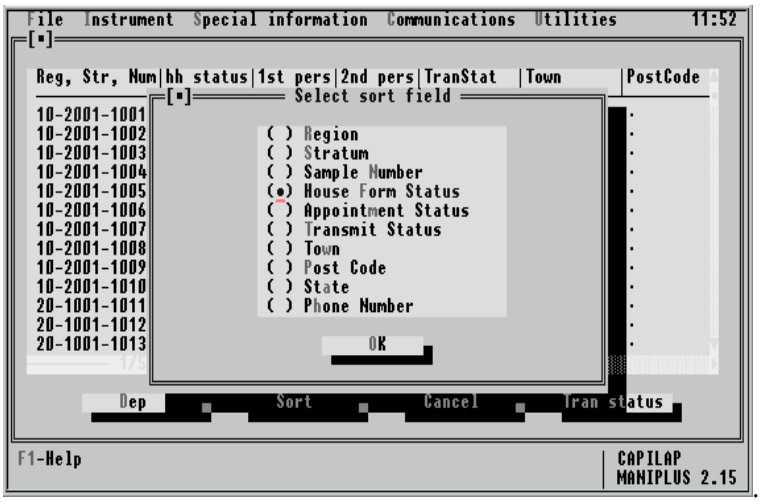
This dialog enables the interviewer to select kinds of forms to work on. For example, the interviewer may wish to review forms in a certain town, to look at appointments, started forms, completed forms, all forms, and so on. Once a choice is made, an interactive lookup table is displayed. The interviewer can scroll up, down, right, and left, sort the list of forms, invoke the instrument for a form, or change its transmission status. In fact, this lookup dialog is the heart of this laptop management system.

Figure 7.3.2: Lookup with all forms selected



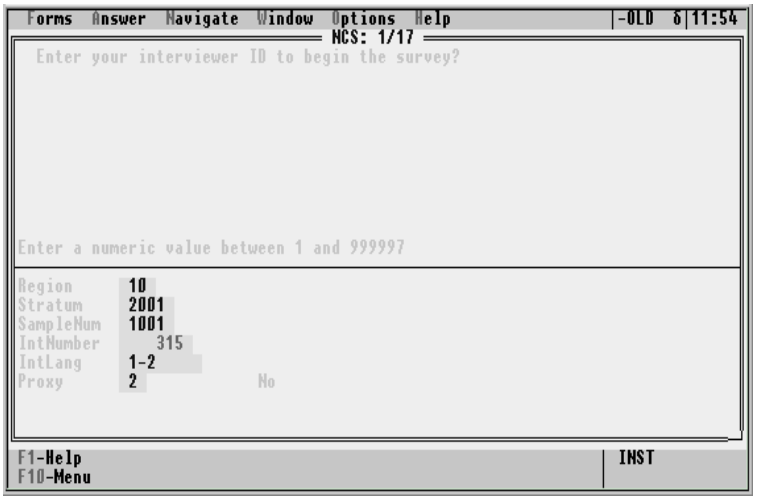
In the figure above, information for all forms is displayed. The interviewer's data set has 50 forms in it. The list is displayed in ID number order when it is first generated. To see the list displayed in a different order the interviewer presses the *Sort* button. In this example, 10 sort options are allowed. Whether the sort order is ascending or descending for each option is in the hands of the programmer. The figure below sorts the table in the order of the household status.

Figure 7.3.3: The select sort field dialog



To start an interview for a particular form, the interviewer scrolls to the appropriate line in the lookup and presses <Enter> or the *Dep* button.

Figure 7.3.4: The DEP interview



To change a transmission status for a form, the *Tran status* button is used. This feature is necessary if the interviewer wants to send a form to the home office that is not complete.

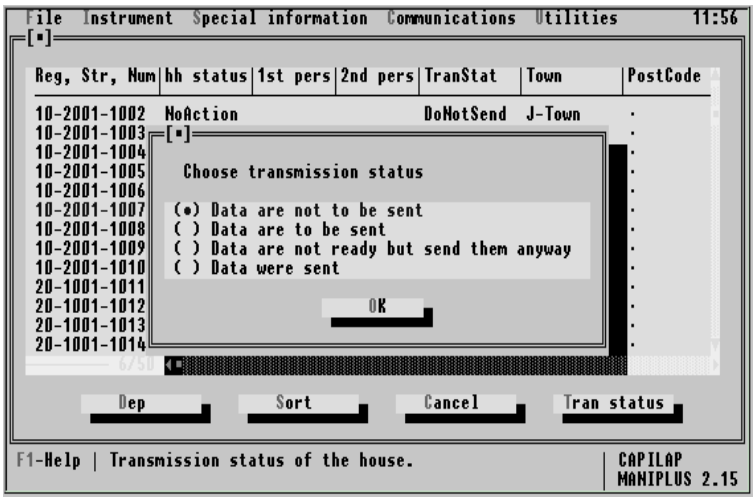


Figure 7.3.5: The transmission status dialog

By combining all of this functionality into one lookup dialog, the programmer can avoid programming many different dialogs. However, several procedures and dialogs are still required to implement all of this functionality including: *PartialLookup, FillPartialTable, ShowTempTable, ChooseFormType, WhichRegion, WhichTown, WhichPostCode,* and *FillTempData*. The procedure that starts it all off is *PartialLookup*. This procedure invokes the other procedures and dialogs depending on the actions of the interviewer.

Enter Known ID

If the interviewer knows an ID number and wants to access just one form, then this dialog can be used. The interviewer has the opportunity to enter the three part ID number in the dialog. The dialog *EnterID* has the setting *VALIDATEDIRECT = NO*. This allows the interviewer to back up and correct a number in the dialog. This also delays the validation of the control cells until they have all been entered, however, the dialog returns the interviewer to the appropriate cell to correct the entry. For most interviewer interfaces, *VALIDATEDIRECT = NO* is appropriate.

Some organisations will prefer to use one control to enter the whole ID string. This is possible with a string auxfield of

sufficient length. A procedure to check the validity of the strung number will have to be written.

It is possible to enter DEP directly and let the interviewer enter the known ID there. This is usually a bit more awkward than the other options, but it is implemented in the next menu choice.

New Form

In some studies an interviewer is allowed to create a new form. This has been allowed here in a naive way. The interviewer is allowed to enter any new ID as long as it doesn't conflict with an existing ID. Usually the laptop management system will determine the new ID number according to a rigid criterion.

Appointments

Appointments deserve special treatment in their own dialog because of the design of the example instrument. For each form there can be either a household appointment or up to two respondent appointments. A separate lookup dialog for appointments is also advisable because the selection and sort choices may differ from those of the lookup dialog mentioned above. In the example below, there are 7 appointments for this interviewer's data base. Two of them are for ID 10-2001-1001.

Figure 7.3.6: The appointment lookup

There are 4 types of appointments used in the example under the heading *SortApp*, an appointment for date and time (in the

lookup as *ADate*), an appointment for a day of the week (*ADay*), a period of time (*APeriod*), and no preference (*ANoPref*). Where appropriate, columns for starting date, starting time, ending date, ending time, and weekday are filled in. The interviewer can sort the appointment list by any of these categories and others as well.

More information for the interviewer is available, either by invoking a dialog or by scrolling. When the cursor is on a form in the lookup table the interviewer can press the button *More Info* to pop up a dialog. This dialog has further information on the appointment for the chosen form only.

Figure 7.3.7: The more info dialog

Another way to see more information for all forms in the lookup dialog is to scroll right with the right arrow. Three more columns of information on who made the appointment, the town of the appointment, and the state of the appointment are shown. The interviewer can scroll right quickly with Ctrl-right arrow and back quickly again with Ctrl-left arrow.

It may be helpful to maintain some appointment information in an external file in order to avoid scheduling conflicts. For example, it may be advisable to allow the interviewer to block

off times for personal schedules and to have one place for all current appointments. This external file and data model can be used by either the main survey instrument or the laptop management system. When the interviewer tries to make an appointment, either program can advise the interviewer of possible conflicts. This kind of functionality will take a lot of thought and experience to specify properly though the programming should not be too hard.

Dialogs and procedures for the appointment lookup table are in the included file *LOOKAPPT.INC*.

Practice

Interviewers often need further practice to become fluent in the use of the Blaise system itself or the main survey instrument. The sub-menu choice *Practice General Blaise* leads to general training modules for different aspects of Blaise, such as reading and answering questions, navigation, or the coding utility. The choice *Practice Survey instrument* allows the interviewer to practice with the real instrument on a practice data set. The choice *Refresh practice forms* allows the interviewer to generate more practice forms in the practice data set.

7.3.3 Special information

The main menu choice *Special information* leads to special kinds of messages and reports that are either supplied by the home office or that the interviewer can generate at any time. These reports are *Special messages*, *Remarks*, and *Frequently Asked Questions*.

Special messages

This invokes a lookup on a file called *MESSAGE* in the subdirectory *EXTERN* that contains special messages from the home office to all interviewers. This can be a way to remind interviewers to perform a special task such as mailing in paper materials. To make sure that interviewers periodically review

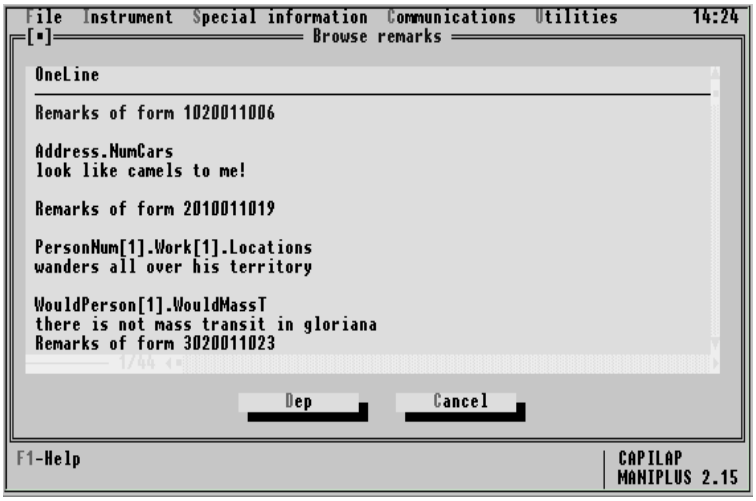
special messages, it is possible to display a dialog upon entering the management system that tells the interviewer when a new special message has arrived.

Remarks

The interviewer may make many remarks while interviewing respondents. Often the interviewer may want to read all remarks that have been made for all forms. The *Remarks* choice allows the interviewer to generate a remarks report and view the report. All remarks made by the interviewer in the laptop's data base are read out into the report.

The sub-menu choice *Read Out Remarks* allows the interviewer to generate the report. It is invoked by the Manipula set-up *READRMK.MAN* in the subdirectory *EXTERN*. The choice *View remarks* allows the interviewer to see the remarks as shown below. This is a lookup on the file *REMARKS.TXT* in the subdirectory *EXTERN*.

Figure 7.3.8: The remarks lookup table

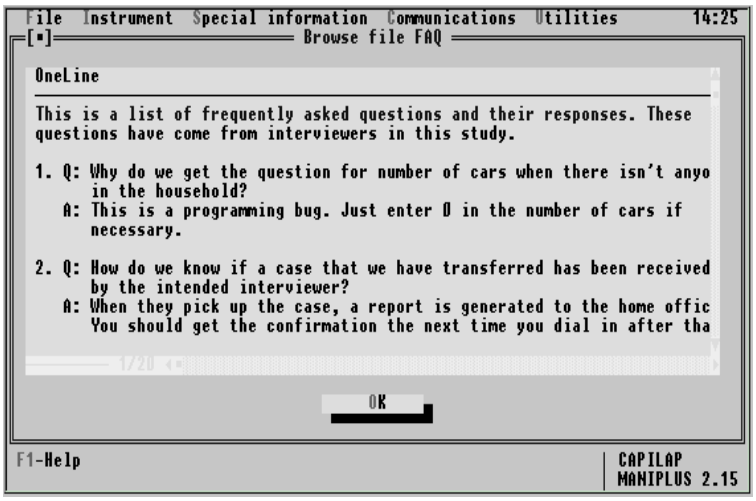


The *View Remarks* choice has been constructed so that if the cursor is on a line with the ID number, the form the remark belongs to can be invoked in DEP.

Frequently Asked Questions

The choice *Frequently Asked Questions* is a report provided by the home office. It can be a compendium of questions that several interviewers have asked. By providing some answers to common questions here, it may be possible to reduce the expense of supporting the interviewers in the field by avoiding unnecessary calls for easy to answer questions.

Figure 7.3.9: Frequently asked questions



This is a lookup dialog on the file FAQ in the subdirectory *EXTERN*.

Other special reports

Other special reports that can be provided to the interviewer (and are not provided here) may include reports on system terminology (for example what the term *page* means in Blaise) or a list of phone numbers to call for various kinds of problems.

7.3.4 Communications

The main menu choice *Communications* is for interviewers to send and receive data and messages to and from the home office.

The sub-menu choices are *E-mail messages*, *Readout forms*, *Dial home*, and *Hot transmit.*

E-mail messages

This is a menu choice that leads to a third party electronic mail system. See Chapter 4, Invoking Programs, on how to invoke a DOS program from Maniplus. An e-mail system can allow the interviewers to communicate special messages to the home office or their supervisors and to receive them. The transmission of e-mail messages occurs when the interviewer dials the home office.

Readout Forms

This menu choice reads out forms from the interviewer's data base that are ready for transmission. These are forms that either have been completed or have been designated by the interviewer to send to the home office despite incompleteness in the lookup dialog under the sub-menu choice *Sort and Select ID* under the main menu choice *Instrument*. The read out forms are placed in their own Blaise data base in the directory *TRANSMIT*. The transmission routine will look for the data base in that directory. The Manipula set-up that reads out the forms is called *SELECT3.MAN* in the same subdirectory.

Forms are read out of the main data set if they have the transmission status *Send* or *SendAnyWay*. Once a form is read out, it is necessary at some point to update the status to *Sent*. This can be done at the time of data read out or later, after some confirmation of successful transmission has been received. The file *SELECT.MAN* does it the first way, the files *SELECT3.MAN* and *UPDATE3.MAN* combine to do it the second way.

Dial home

The menu choice *Dial home* invokes a third party DOS-based telecommunications software and related programs that use the laptop modem and telephone lines to transfer data and e-mail messages to and from the home office. Telecommunications is a very specialised field beyond the scope of this manual. However, there are several things that may happen during data transmission that can be mentioned here. When the interviewer invokes this menu choice, a program (in Maniplus or called by Maniplus) checks for the existence of several files. It may compress them separately or together. Then the telecommunications package is invoked that initialises the

modem and tries to make a connection to the home office. Data may be backed up on diskette as a part of this process. If the connection is made, the data are transmitted to the home office, and any new assignments, programs, or messages are picked up and placed in a special receiving directory. After the connection is ended, a special program supplied by the home office installs the new programs, assignments, and messages in the proper files and directories.

PKZIP® and PKUNZIP®

The file compression and decompression utilities licensed by *PKWARE*®, Inc. can be used directly from Maniplus and Manipula but only if the special parameter **-)** is used. For example:

```
Reslt := RUN('pkzip -) backdat1.zip data.~*')
```

The **-)** parameter must be the first one listed in the command line if there is more than one used. If this parameter is not used, then Maniplus will fail to execute properly and the data file may be damaged.

When files are sent from the office to a laptop, a program must be written to install the files in the proper directories. This is the job of the programming team. Usually this is done as a part of the telecommunications process. It is necessary to completely verify that the installation procedure will work. It is very difficult to fix problems on several hundred laptops.

Hot transmit

The option *Hot transmit* is provided to the interviewer under special circumstances. For example, there may be problem with the data base and the home office wants to look at the whole thing. This choice allows them to do it, but only if the interviewer is given a password. Otherwise the choice is inaccessible to the interviewer.

7.3.5 Utilities

The main menu choice *Utilities* are used only under special circumstances, for example when the interviewer is on the phone to a systems person back in the home office. Both of the sub-menu choices *Monitor Data Base* and *DOS program* are accessible only through a password.

Monitor Data Base

The Blaise utility Monitor can check the integrity of a file, and if necessary, usually rebuild it. If the home office suspects that there is a problem with the data base on the laptop then someone can talk the interviewer through the steps needed to run Monitor. This utility is documented in the Developer's Guide, Chapter 10, section 5. It is usually possible to fix such problems on the laptop in a few minutes and to save the expense of transmitting the whole data base home for repair.

Problems with the data base may occur if the interviewer turns off the machine or the battery fails at the wrong time. The data base is especially designed so that the main data file and the index files can be regenerated with the Monitor utility. The best policy is to educate the interviewers in the proper use of the laptop computer, including how to shut it off and battery management.

DOS program

With this menu choice, which is password protected, the home office can instruct the interviewer to run DOS commands or BAT file programs to try to help diagnose problems. Again, the purpose here is to resolve problems on the laptop and to save the expense of exporting the whole data base, or the machine itself, to the home office and to avoid down time for the interviewer.

7.4 Making it easy for the interviewers

The two most important things you can do to make it easy for the interviewer to use the laptop management system well are: design it well in the first place (keep it as simple as possible) and, train the interviewers well in the use of it. There are three more things you can do to make life easier for the interviewer to use the laptop system. You can provide hot keys for the most common menu choices, you can provide a help facility, and you can provide the interviewer with ways to back out of dialogs or to back up in a dialog. All three techniques have been implemented in this example.

Hot keys

Hot keys are function keys that the interviewer can press in order to go quickly to a dialog without having to navigate the menus. Chapter 2 on menus explains how to assign hot keys. In the example laptop management system, the hot key assignments have been chosen so that they do not conflict with function key assignments for the Data Entry Program. This is not a Blaise requirement, but is done so as to not confuse the interviewers. The function keys used in the DEP are as follows:

Function key	**Function**
F2	Calculator
F3	Next form language
F4	Errors in field
F5	Don’t Know
F6	Refusal
F7	Save and continue
F8	Search tag
F9	Make remark

The function keys used as hot keys in the laptop management system all use the *Shift* key. The hot keys are as follows:

Function key	Function
Shift F1	Reasons to participate
Shift F2	Sort and select ID
Shift F3	Appointments
Shift F4	Special messages
Shift F5	Frequently Asked Questions
Shift F6	Generate remarks
Shift F7	View remarks
F10	Main menu

Only commonly used dialogs are provided with hot keys. Some functions, such as reading out data, are not given a hot key since you wouldn't want the interviewer to start such a process accidentally, even though a confirmation dialog is provided.

When you provide hot keys, it is necessary to additionally use a *TEXT* dialog element in the dialog to label the dialog. Otherwise, the interviewer might be confused as to which dialog has actually been invoked. See the dialogs *ChooseFormType*, and *ChooseAptType* in the source code and press *ShiftF1* and *ShiftF2* in the laptop management system for examples of this.

Help facility

Chapter 4 describes how to implement the help facility. There is extensive help available in this example that is available for all menu selections and most dialogs with the F1 key. Additionally, the help facility is used to provide a way for interviewers to give reasons to respondents on why they should participate. The ASCII version of the help file is *CAPILAP.TXT*.

Backing out and backing up

Interviewers will make wrong menu choices or enter wrong numbers in dialogs. There are three things you can do make it possible for interviewers to back out of a dialog or back up in a dialog. They are: the *CONFIRM* function, the *ESCAPE = YES* setting, and the *VALIDATEDIRECT = NO* setting. The latter has been covered above already.

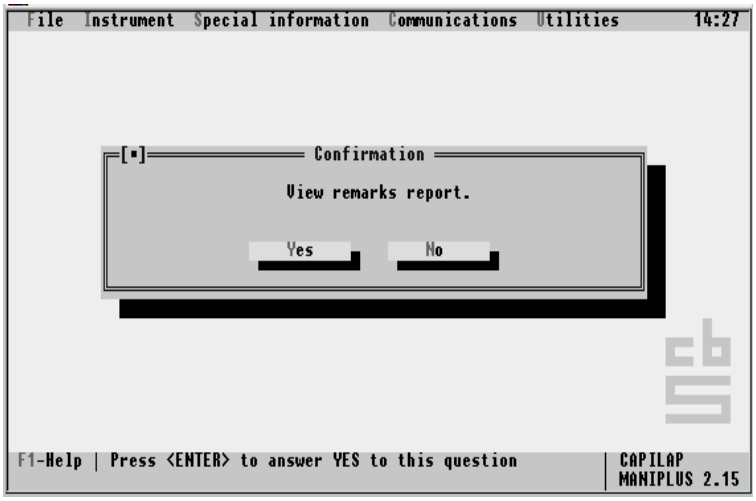
CONFIRM

The *CONFIRM* function displays a dialog with the buttons *Yes* and *No* and displays a message that the programmer has provided. For example, the following code:

```
IF CONFIRM('View remarks report.') THEN
```

produces the following dialog when the interviewer has chosen the menu selection to view remarks.

Figure 7.4.1: The confirm dialog



ESCAPE = YES

The dialog setting *ESCAPE = YES*, which is the default setting for dialogs, allows the interviewer to escape the dialog without having to execute an action. There are places where *ESCAPE = NO* is appropriate, but for almost all dialogs the interviewer should have the possibility to escape.

VALIDATEDIRECT = NO

This dialog setting is explained above. With this setting, the interviewer can back up in the dialog if a wrong number has been entered. If *VALIDATEDIRECT = YES*, the interviewer cannot back up in a dialog until all controls have been filled in.

7.5 Building a multi-survey laptop management system

It is possible to build a multi-survey laptop management system in Maniplus, and this has been done by some organisations. The construction of such an example is beyond the scope of this chapter, however, some considerations can be mentioned.

Master Maniplus set-up

A master Maniplus set-up can be programmed to allow the interviewer to choose which survey to work on. In its simplest manifestation, the master set-up can be a simple menu system with a current list of surveys. When a survey is chosen by the interviewer, a subsidiary Maniplus set-up can be invoked that loads the appropriate instruments for that survey. Each survey would have its own subsidiary Maniplus set-up that is invoked with the *CALL* feature. The instrument for the survey would not be loaded until the survey is chosen and the subsidiary Maniplus set-up is called.

Cross survey management

A key advantage of a multi-survey laptop management system is to allow the interviewer to manage the workload across several surveys. Thus some of the functionality included in the example of this chapter would have to be moved up to the level of the master Maniplus set-up. This may especially apply to features such as review of appointments over all surveys, review of respondent locations over all surveys, special messages from the home office, and telecommunications.

Administrative data

In a multi-survey management system, the location of the administrative and management data including appointments should be in external files. This makes it easier to generate cross-survey reports.There are many other considerations that come into play when designing a multi-survey management system. Only a few can be mentioned here, others are very specific to the organisation.

8. Top-Down Tabular Editing

Blaise is well known as an interactive editing tool, used after data are collected, either on paper or by electronic means. This kind of data editing is known as form-by-form editing. In traditional form-by-form editing, the data editor works with each form until it has been cleaned of soft and hard errors. She repeats the process for a whole data set until all forms are clean. Blaise as an interactive editing tool is extremely efficient when compared to old-style batch methods with errors printed out on pages of computer sheets. However, there are some drawbacks to form-by-form editing.

- In almost all statistical organisations, it is necessary to clean all forms in the data set before the data are considered sufficiently clean for tabulation or summary.
- The data editor may spend a lot of time working on forms and edits that have no real effect on the estimates.
- Despite all the effort, some data problems could still slip through, revealing themselves only very late in the process (e.g., at summary) when it is very difficult and expensive to correct them.
- Even with the considerable power present in Blaise interactive editing, some data editors find traditional form-by-form data editing very tedious.
- Soft edits are often set too tightly and too frequently, at times with no regard to data distributions, resulting in much work to suppress edits.
- Where there are many edit flags, or in the case of Blaise, edit signals, it can be hard to spot the important problems in the forest of unimportant problems.

For these reasons and because computing advances allow it, another kind of editing, called *top-down editing* has been

developed. In top-down editing, the data editor views the data at an aggregate (or top) level, detects problems there, then ‘burrows down’ to the offending record or records.

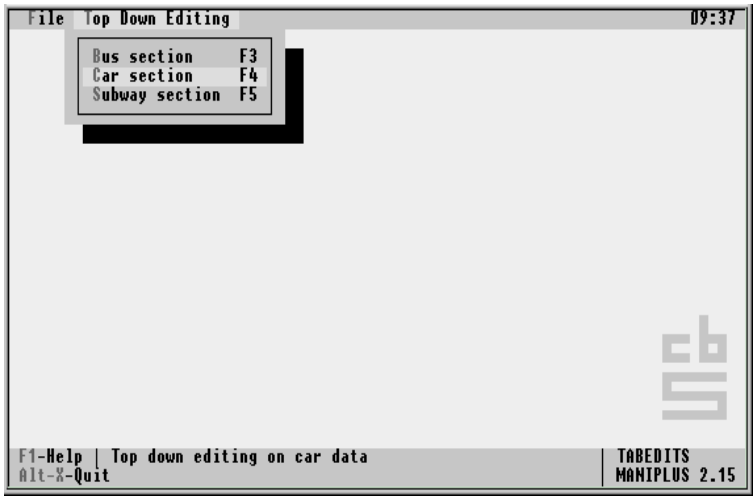
In Maniplus, it is possible to implement a kind of top-down editing called top-down tabular editing. The programming required is sophisticated and it is necessary to design lookups used for top-down editing in advance. If several lookups are planned for a large data file, the developer must consider how this is best implemented in order to generate the lookup tables most efficiently. Especially, it may be necessary to include a generic secondary key in the data model that can be quickly updated in order to efficiently generate tables with for different parts of the data base. If a Blaise data file is being used in CATI, or in interactive editing, there may be a tension between generating timely lookup tables and complete lookup tables. This latter issue is handled with ONLOCK = CONTINUE or ONLOCK = WAIT It is up to the developer which method to use.

A good reference on top-down editing, also called macro-editing, is found in Granquist (1995), who has been one of the pioneers and leading proponents of this important idea.

8.1 Example Maniplus set-up for top-down editing

An example instrument called *TABEDITS.MAN* is available with the Blaise distribution. It and supporting files are found in *\BLAISE31\DOC\MP_CHAP7*. The *README.TXT* file explains how to set up the example. It concerns a commuting survey where detailed information are collected on use of bus, subway, and car for transportation to and from work. The first screen has the main menu option *Top Down Editing*. When this choice is selected, the editor can choose to edit the bus, car, or subway section.

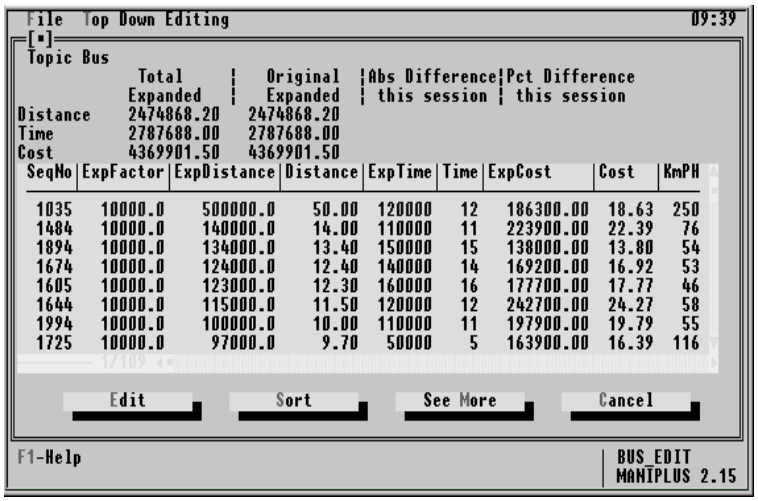
Figure 8.1.1. The Top-Down editing menu



Tabular top-down editing in Maniplus proceeds as follows. It is illustrated with the Bus section (F3).

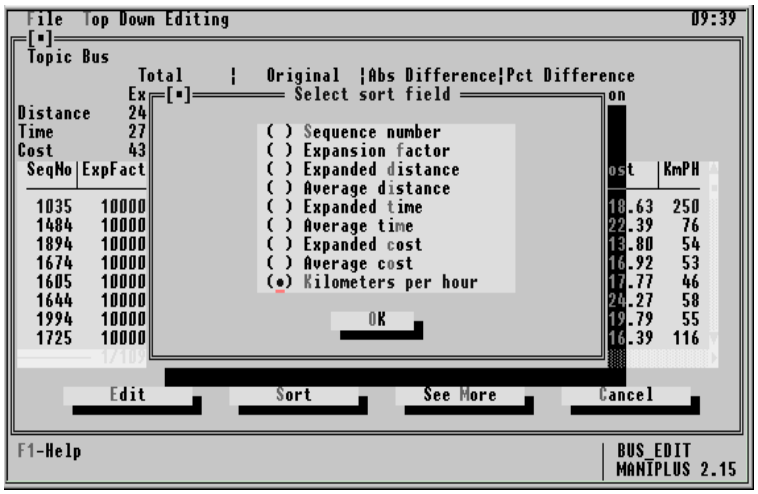
- The data editor browses or inspects the tabular view of data at the top or aggregate level. The editor sees expanded totals and sorted records that are the constituent parts of the expanded total. Several columns of fields are displayed with the ability to sort on any of the fields.

Figure 8.1.2. Table for bus commuting



- The editor has the ability to sort on any of the fields (i.e. columns) in the table.

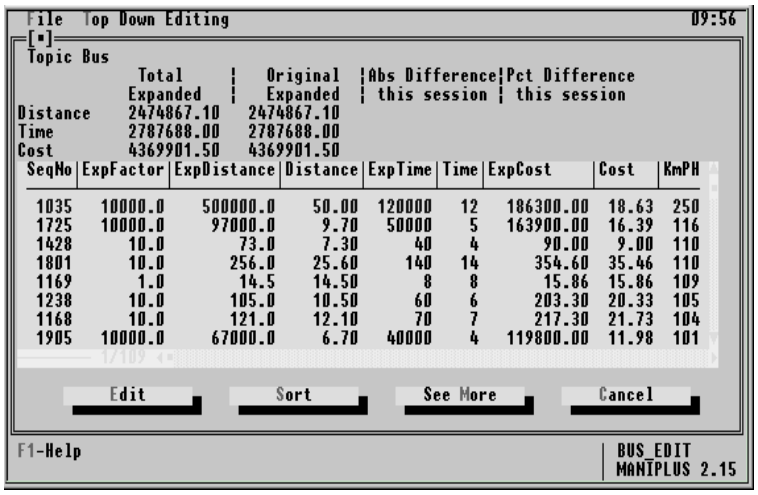
Figure 8.1.3. The sort dialog



- In the interactive table, now that it is sorted in descending order by the right-hand column KmPH, an outlying data item has become evident in record number 1035. The kilometres per hour for this record is twice the rate of the next record in

the list in the order of KmPH. The data editor can choose the record by placing the scroll bar on it.

Figure 8.1.4. Records sorted on KmPH



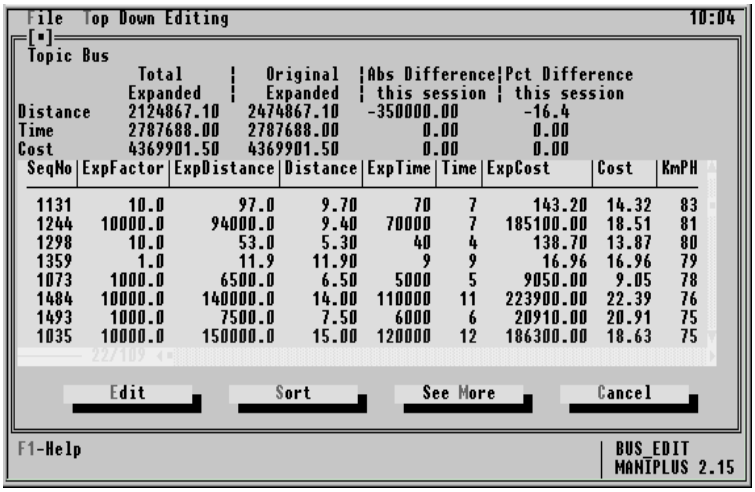
- By pressing Enter, the offending record containing the data item is invoked and brought up in editing mode in the Data Entry Program. At the screen for the bus data, the data editor has changed 50 to 15 for distance, reasoning that the interviewer misheard the response.

Figure 8.1.5. DEP screen for bus editing

- After the data editor has changed 50 to 15 and leaves the Data Entry Program, she finds herself back in the interactive

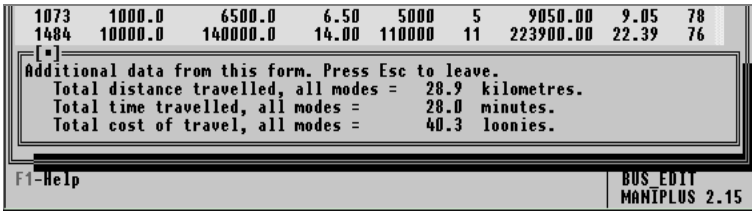
table. Note that now record 1035 is not the top record anymore. This is because the kilometres per hour for this record is now 75 where before it was 250. The cursor has stayed with the record even after the re-sort.

Figure 8.1.6. Changed order after the change



- The summary statistics at the top of the screen show the original expanded total and the current expanded total for distance for the topic bus. It gives the absolute amount of decrease and a percent decrease of the expanded data. Note that this one change has had considerable impact on the expanded data.
- Another option the data editor has is to see more data on the record by pressing the *See More* button. When this button is pressed, a window pops up showing values of selected data items. The data editor leaves this window by pressing *Escape*.

Figure 8.1.7. The more window



8.2 Alternative display options

The example in this chapter could have been programmed differently. For example:

- All expanded numbers could have been left out, leaving only unexpanded numbers and derived fields left in the display. This might satisfy methodologists who worry that subject matter people would 'edit until the expansions turned out right'.
- A block-level audit trail could have been programmed so that if the editor changes any data say in the bus block, both before and after numbers would be kept.
- For periodic surveys, expanded totals from the last period could have been part of the summary display.
- It was not absolutely necessary to bring in all of records containing bus data into the bus interactive table. Some statistical decision technique could have been applied to reduce the number of records available for editing. However, there is not much performance penalty associated with bringing in all records, so it is probably better to have all applicable records present in the interactive edit table.
- Another possible presentation is to have only ratios or rates displayed in the columns, for example, the *KmPH* field. There could also have been a column headed *CostPerKm*, or *TimePerKm*.
- The interactive table could have been much wider. In that case it would have contained more columns (higher dimensionality) and the editor could scroll horizontally.

There are many possibilities. What display is used should be decided with methodologists and the data editors.

8.3 Advantages of top-down editing

There are several advantages of top-down editing.

- It combines editing with summarisation at an early stage, when it is possible to inexpensively change bad data.
- Data editing can be concentrated on relatively few records, just those with the biggest impact on the estimates.
- The data editor can see the effects of the changes on the expanded totals.
- The concept of a clean data set is changed. In top-down editing, a ‘clean enough’ data set may be sufficient. In particular, the organisation may stop editing every record until they are all clean. They may decide to edit until no discernible changes at the top level can be seen. This may be done by treating a small fraction of the data records.
- Since this kind of editing is cross-form or cross-record, the data set itself may not be ‘clean’ at all, but the fields in the view may have reasonable values when expanded and aggregated. Thus it is possible to publish estimates from a section of the survey without having to clean the whole data set.

8.4 Advantages of a tabular display.

Another top-down editing technique uses graphical displays of as the overview that is presented to data editors. For example, there can be a scatter plot where the data editor invokes a record by clicking on a data point with a mouse. Maniplus cannot yet do this, except perhaps with the help with DLLs. While graphical top-down editing has its place, there are a few advantages of tabular top-down editing as done in Maniplus.

- The main output of most statistical organisations is tabular. Graphs and charts are published too, but it is the numbers that are used by other people with their software. If the aggregated numbers are the primary output, then it is better to edit tables, not charts.
- It is easy to show many dimensions in tables. The table shown in this example has three basic items, each displayed as a number and as an expanded number (*Distance*, *Time*, and *Cost*). It would have been just as easy to display several more columns on different questions.
- As implemented here, it is easy to see the results of editing on the expanded data.

8.5 Methodological concerns

There can be abuses of top-down editing. Primarily, it is possible for a data editor to ‘cook’ the data and to ‘set’ estimates at desired levels and to make data look plausible all the way down to the record level. Nevertheless, the technique of top-down editing should and will gain further use because of its ability to cut survey costs by vastly reducing the amount of time (which can be great) in post collection review. The challenge to the statistical organisation is to implement top-down data editing with proper (not onerous) controls so that the positive aspects can be realised without any of the negative possibilities. Several ways of controlling the technique are possible. Some are:

- Create an audit trail of every block of data that has been invoked and changed.
- Where a data item has been changed, ask for a reason for the change. This can be done with dialogs upon leaving the Data Entry Program.
- Constrain the data editor’s view of the expanded totals to part of the overall data set, for example to a region, to a commodity, or to an industry group.
- Eliminate displays of expanded data.

While there are ways to control abuse, it is not necessarily a good idea to implement all (sometimes expensive) controls that can be imagined. The end result may be to irritate data editors while not really controlling the problem.

8.6 Top-down editing on a Local Area Network with two or more people

In regular interactive editing, the unit of editing is the form. On a Local Area Network, either a *CATI* interviewer has possession of a form, a data editor has possession of a form, or no one has possession of a form. If someone tries to access a form that is already on another workstation, a nice message is displayed informing him of that fact. The editor can choose another record to edit and not much time has been lost.

With top-down editing, the unit of editing is a slice of data across all records. A potential network conflict is with the generation of the interactive top-down table. When the top-down table is generated, it goes through all forms in the data set, to find those with applicable data, and displays them. Suppose another data editor has invoked the Data Entry Program on a form in the data set. The conflict is that Maniplus does not have access to the record that is being used on the other workstation. Maniplus can wait (*ONLOCK = WAIT*) for the record to be released by the other data editor, but maybe he has gone to lunch. Or it can bypass that record, (*ONLOCK = CONTINUE*) but then maybe that record contains an important outlier. It is important to realise that this problem exists only if a record has been brought up in the Data Entry Program in another work station at the time of generating the interactive table.

Each organisation must decide how to cope with this potential conflict. For example the concerned individuals could co-operate on the generation of the top-down tables. The other top-down editor could be asked to get out of the Data Entry Program for a few minutes, however, he could leave the interactive table on his screen. Or the tables could be generated at night, ready for the next morning.

8.7 Combining top-down and form-by-form editing

The example in this chapter effectively combines top-down editing with form-by-form editing. By using top-down techniques, the data editor can focus her energies on important problems. Once she has invoked a record and has entered the Data Entry Program, however, she is able to edit the whole form just as if she invoked the form by other means. The advantage of this is that if she changes a data item in the data entry program she can see immediately if she has violated any other edits. Especially important is whether or not she has violated any hard edits in the form. If so, she can repair the form as needed until no hard edits are left. She is then assured that the integrity of the data has been maintained.

Once in the Data Entry Program, the data editor should not worry too much about soft edits. The important ones can be found through the top-down view.

Form-by-form data editing is a very important technique in its proper place. That proper place is in the interview, where data problems can be cleared up with the respondent. After data are collected, the technique of top-down editing should be used as much as possible to diminish the costs of the survey. Who knows, you might be able to get a promotion by finding ways to increase sample size in an age of diminishing budgets.

An efficient way to collect and process data is to use Blaise for data collection, Blaise interactive form-by-form review to clean up only hard edits, and top-down editing in Maniplus to review selected soft edits. Pierzchala (1996) discusses where to implement each kind of edit, and the challenges in doing so. Granquist (1995) also looks at the role of editing at various stages in the statistical production process.

8.8 Programming

Most programming techniques needed to produce this example have been covered elsewhere. Programmers comments have been made in the source code of the example set-ups. Some important points unique to this example are discussed briefly.

8.8.1 Use of a master file

The master file *TABEDITS.MAN* loads the data model *COMTEL.BLA*, implement a menu system, and calls subsidiary Maniplus set-ups. In other words, it organises the top-down editing and expedites processing by loading the data model.

8.8.2 USES section in master and subsidiary files

The *USES* section of *TABEDITS.MAN*, *BUS_EDIT.MAN*, *CAR_EDIT.MAN*, and *SUB_EDIT.MAN* all mention the data model *COMTEL.BLA*. This is necessary for syntactical reasons. For example, you can invoke *BUS_EDIT.MAN* by itself (not from the master set-up *TABEDITS.MAN*) and it would work. However, the data model is loaded only once into memory if a master set-up and a subsidiary set-up both use the same data model.

8.8.3 One subsidiary set-up for each top-down table

There is one subsidiary set-up for each top-down table. This is necessary where a secondary key and filters are used. For example, from *BUS_EDIT.MAN*:

```
UPDATEFILE ComData : INST ('Comtel', BLAISE)
  SETTINGS
    OPEN = NO
    CONNECT = NO
    KEY = SECONDARY(Use_Of_Bus)     {unique to setup}
    ACCESS = SHARED
    ONLOCK = WAIT
  FILTER
    SeqNo
    Weight
    TotDistance
    TotTime
    TotCost
    BusPresence                     {unique to setup}
    Bus                             {unique to setup}
```

Even though a separate set-up is used for each top-down table, the use of *INCLUDE* statements reduces the code needed. The following are the files included in each of the set-ups *BUS_EDIT.MAN*, *CAR_EDIT.MAN*, and *SUB_EDIT.MAN*.

```
MenuDesc.Inc
TempData.Inc
TempFile.Inc
AuxField.Inc
Fillsort.Inc
EditList.Inc
```

The reason that so many files could be included between the three set-ups is that each of the topic sections in *COMTEL.BLA* are repeated instances of the same block. If another block structure was involved, it would have been necessary to adapt much code to the different block.

Secondary keys are used in *COMTEL.BLA*, one for each topic. They are: *Use_of_Car*, *Use_of_Subway*, and *Use_of_Bus*. These secondary keys speed up the generation of the interactive top-

down editing table. For example, when bus as a topic is chosen, the set-up *BUS_EDIT.MAN* looks only at those records with a positive secondary key for bus presence. In the example, there are approximately 100 bus records in a data set of 1000 records. Only those 100 records are visited by the set-up.

However, it is not a good idea to have a lot of secondary keys in a data model. They take up space, and if there is ever a need to regenerate the data set with Monitor, it can take a lot of time if there are many. This presents a challenge in a large data model where there are many sections, each worthy of top-down editing, and where there are many records. One way around this is to have a generic *STRING* field in the data model which is a secondary key. This secondary key field can be computed quickly with a small Manipula set-up to reflect the presence of data in a particular section of the data model. Then the top-down table could be generated quickly resulting in an overall savings of time.

8.9 References

Granquist, L. (1995), *Improving the Traditional Editing Process*, in Business Survey Methods, Chapter 21, pp 385-402, Wiley Interscience, Cox, Binder, Chinnappa, Christianson, Colledge, and Kott editors.

Pierzchala, M. (1996), *CAI and Interactive Editing in One System for a Survey in a Multimode Environment*, in Data Editing Workshop and Exposition, Statistical Policy Working Paper No 25, Office of Management and Budget, Washington, DC, Alvey and Jamerson editors.

9. Enhanced CATI management

Blaise offers a full *CATI* survey management capability that starts with the call scheduler and call management system which come with the basic system. Together they offer a solid core of call management features. They are completely flexible in the handling of survey definition, calling times, busy signals, appointments, time zones, groups of interviewers, quotas, and other features. All of these features are easily controlled through on-line entries in dialogs in the call management system. Additionally the call management system provides a wide variety of reports that can be viewed from the management system dialogs or printed to file. None of this core capability has to be programmed, they are inherent features of the Blaise system. As such it is possible to implement many *CATI* surveys in very little time. Additionally it is useful to have a complete understanding of how each of the call management system features can be used to provide some of the enhanced management capabilities. For more information about the Blaise call management system see chapters 11 and 12 in the Blaise Developer's Guide.

For more enhanced call management capability, you may need to use one or more of other tools including: specialised parallel blocks in the data model, Manipula (data manipulation utility), and Maniplus. This chapter will discuss briefly how these tools combine to provide an enhanced *CATI* management solution, concentrating on the use of Maniplus. A complete treatment of the call management capabilities of Blaise is beyond the scope of this chapter, as it would take several hundred pages. However it will suggest ways that you can proceed with specialised *CATI* management needs.

9.1.1 User-control of call management

Some organisations customise call scheduling and call management in order to increase contact and response rates and lower costs through increased productivity. Enhanced *CATI* call management includes (but certainly is not limited to):

- execution of user-defined *strategies* and *protocols* to handle the current and future scheduling of no-answer, answering machine/service, and other call outcomes based on the call history of, or other information, about each number,
- special handling of non-response, especially the classification of refusals into soft and hard refusals with future attempts being made on soft refusals,
- special handling of missed appointments,
- changing the priority of a form,
- changing the group or person assigned to handle a form,
- production of specialised reports according to client or organisation wishes,
- utilising an elaborate user-defined call outcome coding scheme that influences or determines how the call scheduler operates,
- spawning a new form under tightly controlled circumstances,
- scheduling cases meant for groups or individuals for times when the group or individual will be available,
- using the call scheduler and call management system to implement sophisticated sample designs with numbers being delivered to interviewers based on call outcomes of previous calls,
- sharing management information between in-field *CAPI* laptop computers and the in-office *CATI* call management information when these two modes of data collection are operating on the same survey at the same time, and

- removing cases from the call management system without interrupting the calling, where paper questionnaires are received in the mail.

9.1.2 Terminology and concepts

Much of the literature on *CATI* management make use of terms such as *cases*, *strategies*, *queues*, *time-slices*, and so on. The Blaise system does not make use of these terms. Nevertheless, all of the functionality implied by these terms is available within the Blaise system. This is not the place to give a complete glossary of *CATI* terminology, but rather to give some illustrative examples of how some *CATI* concepts can be implemented in Blaise.

Case

The term *case* which is standard usage in *CATI* systems is synonymous with the term *form* in the Blaise system. The term *case* is used in this chapter.

Time slices

An example of a time slice is Thursday between 2:00 and 4:00 PM, or 2:00 to 4:00 PM Monday through Friday. In the Blaise appointment scheme the *period appointment* can be used as a time slice among other things.

Strategies

An example of a *strategy* is in the case of a no-answer call outcome. If a call is made on a weekday evening and there is a no-answer outcome, it may be best to make the next call for that phone number on a weekend. In Blaise, you would use Manipula or Maniplus to reschedule the form for the desired day, according to whatever algorithm or rescheduling scheme the organisation cares to program.

Queues

An example of a *queue* is assigning a form to be handled by certain interviewers. In Blaise this is done with groups of interviewers. There are variations on this theme. Blaise can handle these too.

Where many other call management systems are based on an explicit queuing system, Blaise *CATI* call management is based on a file called a *day batch* which is created daily. Queues as a concept, rather than as a formal device, are implemented within the workings of the Blaise call management system and how it operates with the day batch. While the terminology and architecture of the Blaise call management system may be different from other systems, enhanced *CATI* management functionality is present, and very powerfully so.

9.2 Example instruments and Maniplus setups

There are two example instruments and setups used in this chapter. They are *OEPS.BLA* and *OEPSPLUS.MAN* found in *\BLAISE31\DEMOS\CATI*, and *COMTEL.BLA* and *HISTORY2.MAN* found in *\BLAISE31\DOC\MP_CHAP9*. In these directories there are several other files that help to setup the examples. These are:

```
COMTEL   BLA   {Main survey instrument file.}
HISTORY  BLA   {Description of the history file.}
MYLIB    LIB   {Library of types.}

{Subject matter blocks included with Comtel.bla.}
WORKPLCY INC
ADDRESS  INC
CATI     INC
CATICAPI INC
BPERSONT INC
BWORKT   INC

{Parallel blocks used with the call scheduler and that
 are included in Comtel.bla.}
APPOINT  INC
NONRESP  INC
NOANSWER INC
BUSY     INC
AMACHINE INC
DISCONCT INC
OTHER    INC

COMTEL   OPT   {Btemula option file for CATI emulation.}
COMTEL   S??   {Files of this form are used by
                COMTEL.OPT.}
COMTEL   ~TS   {CATI survey definition file.}
COMTEL   TST   {Test data that are read in with
                READIN.MAN}
READIN   MAN   {Reads test data into Comtel.}
GENDATA  MAN   {Generates test data.}
HISTORY2 MAN   {Maniplus setup that operates on the
                day batch when calling is taking place.}
```

Prepare *MYLIB.LIB*, *COMTEL.BLA*, *HISTORY.BLA*, *READIN.MAN*, and *HISTORY2.MAN* in this order. Then execute *READIN.MAN* in order to create a *COMTEL* Blaise data set. Invoke the *CATI* management

system and confirm that the survey definition file is *COMTEL.~TS*. Change the dates for the valid survey period and create a day batch. The *CATI* survey should be ready to run. If you have the *BTEMULA* utility, you can use the *COMTEL.OPT* file and its associated *COMTEL.S??* files to run a *CATI* emulation on a Local Area Network. If you want to modify the test data set for your own situation, you can modify the Manipula setup *GENDATA.MAN*.

9.3 The role of parallel blocks in call management

It is possible to attach a user-defined parallel block of questions to each call outcome possibility listed on the dial screen that is presented to the interviewer. With this capability you can gather more information about each call outcome and implement your own elaborate call outcome coding scheme. Once defined the parallel blocks are easily reused from survey to survey. One example is the *Other* call outcome category.

```
BLOCK BOtherOutcome
  PARAMETERS
    IMPORT Whom, Phone : STRING
  FIELDS
    Details "What kind of other phone call was this?"
           : (Modem "Computer modem whistling sound",
              PayPhone "Pay phone",
              Mobile "Mobile phone",
              Language "Language problem",
              Message "Pre-recorded messages such as
                       time, weather",
              Wrong "Wrong number",
              Inacc "Inaccessible respondent",
              Silent "Silent line",
              OtherOne "Any other outcome.")
    OtherKind "Explain what other kind of call outcome
               there was." : STRING[20]
    Answer "Did a person answer the phone?" : TYesNo
    Confirm "Were you able to confirm that you had
             dialed correctly?" : TYesNo
    CallBack "Try the phone number one more time. It
              is @Y^Phone@Y." : TContinue
    Lingo "Which language does the respondent speak?
         @/@/@Y[Int] Don't Know allowed.@Y" : TIdiom, DK
    OtherLingo "Can you describe the other language?"
               : STRING[20]
```

```
  RULES
    Details
    IF Details = OtherOne "" THEN
      OtherKind
    ENDIF
    IF Details = Language "" THEN
      Lingo
      IF (Lingo = OtherIdiom) OR (Lingo = DK) "" THEN
        OtherLingo
      ENDIF
    ENDIF
    IF Details <> Language "" THEN
      Answer
    ENDIF
    IF Answer = Yes THEN
      Confirm
    ENDIF
    IF (Answer = No) OR (Confirm = No) THEN
      CallBack
    ENDIF
ENDBLOCK
FIELDS
  OtherOutcome : BOtherOutcome
```

When the interviewer selects the *Other* call outcome category the questions above will be asked before the case is left. With an enhanced set of pre-defined outcome codes such as the above, it is possible for a Manipula or Maniplus program to reschedule the interview and assign it to the correct person. For example, suppose an English speaking interviewer encounters a French speaking person on the line. If the interviewer does not speak French well enough to conduct the interview, she can choose *Other* and record that there was a language problem and that the respondent speaks French. The case can be re-assigned to a time when a French speaking interviewer will be available.

Other outcomes, such as a hostile refusal, can be sent to the supervisor for a determination of what to do with the case.

9.4 The role of Manipula and Maniplus in call management

Manipula and Maniplus both have a role to play in enhancing the call management system. Manipula is used when calling is not taking place, perhaps after calling is done for the night. Most enhanced *CATI* call management can be accomplished with Manipula which comes with the basic system. Maniplus can be used both during and after calling.

Manipula does not directly modify the day batch. It can be used to make computations in the Blaise data set fields such as appointments and target interviewer. When the day batch is created, the *CATI* management system takes into account these values and includes cases into the day batch as is appropriate.

Maniplus can also make computations into the Blaise data set. Additionally it can directly affect the day batch while it is being used by the call scheduler. This means that real-time modifications can be made to the call scheduler.

9.4.1 Manipula

Many requirements listed above (and others) are easily met by using the full range of options in the call management system by itself, or by using Manipula. It can control how the call scheduler works by determining how a file known as a *day batch* is created. For example, Manipula can calculate the values of user-defined management fields in the survey data set such as appointments or other fields which can be designated as *selection fields* or *sort fields*. The call management system takes account of these appointments and fields when it creates the day batch. See chapter 11 of the Blaise Developer's Guide for further information.

Manipula setups that use the day batch cannot be executed during calling. For most needs this is not a limitation since the call scheduler parameters allow you to put aside phone numbers for no-answer and other call outcomes until the end of the day's calling. Then Manipula can be used to re-schedule these forms according to any criteria.

9.4.2 Maniplus

Maniplus can modify the day batch itself while calling is taking place. It does this by deleting or adding forms from the day batch. This is useful for situations where the manipulation of the day batch cannot wait for the day's calling to end, as for example, when late arriving paper questionnaires necessitate the removal of forms from the day batch while calling is taking place. The two key words that add and delete records from the daybatch are *daybatch_add* and *daybatch_del*.

The use of the two key words is illustrated in the example setup supplied with the Blaise distribution called *OEPSPLUS.MAN* found in the *DEMOS\CATI* subdirectory.

```
PROCESS OepsPlus

SETTINGS
 PROGRESS=YES  {Shows progress like regular Manipula.}

USES
  Oeps                        {Main survey instrument.}

  DATAMODEL Adjust            {ASCII transaction file.}
  FIELDS
    ID: 1..999
    InOut: (DB_Add,DB_Remove)
    STime: TIMETYPE
    ETime: TIMETYPE
    Prio: (p_default,p_soft,p_medium,p_hard)
    Who: STRING[15]
  ENDMODEL
```

```
UPDATEFILE OepsData:Oeps ('oeps',BLAISE)
SETTINGS
  ACCESS=SHARED
  ONLOCK=CONTINUE
  CONNECT=NO

INPUTFILE AdjustData:Adjust ('oeps.adj',ASCII)
SETTINGS
  SEPARATOR = ','

MANIPULATE
  WHILE NOT (AdjustData.EOF) DO
    AdjustData.READNEXT
    OepsData.GET(AdjustData.ID)
    IF OepsData.RESULTOK THEN
      IF InOut = DB_Add THEN
        CASE Prio OF
          p_default:
          OepsData.DAYBATCH_ADD(STime,ETime,DEFAULT,Who)
          p_soft:
          OepsData.DAYBATCH_ADD(STime,ETime,SOFT,Who)
          p_medium:
          OepsData.DAYBATCH_ADD(STime,ETime,MEDIUM,Who)
          p_hard:
          OepsData.DAYBATCH_ADD(STime,ETime,HARD,Who)
        ENDCASE
      ELSEIF AdjustData.InOut=DB_Remove THEN
        OepsData.DAYBATCH_DEL
      ENDIF
    ENDIF
  ENDWHILE
```

This setup uses the ASCII transaction file as the driving file within the *WHILE-ENDWHILE* structure. For each record in this ASCII transaction file a record is either added or deleted from the daybatch.

Use of a transaction file

When adding or deleting records to the daybatch, it is not necessary to process the whole daybatch. It is better to process only those parts that are necessary to modify, add, or delete. For many tasks, an excellent ASCII transaction file to use is the *CATI* history file. In this example, it is named *COMTEL.~TH*. An excerpt is shown where the even number lines continue the odd number lines.

```
1349,350,19961113,12:16:26,1,1,Int1,2,3,1,2,,
        12:16:35,9,3
1079,80,19961113,13:29:17,1,1,Int2,2,5,4,1,,
        13:30:24,67,65
1093,94,19961113,13:32:17,1,1,Int3,2,7,4,1,,
        13:32:58,41,39
```

Int1, Int2, and Int3 stand in for real interviewer names. This file is described by the data model *HISTORY.BLA*. Part is shown.

```
FIELDS
    {The actual length of the primary key depends on the
     questionnaire.}
    ThePrimKey : INTEGER[4], EMPTY
    InternalKey : INTEGER[8]
    DialDate : DATETYPE         {date format = YYMMDD}
    DialTime: TIMETYPE
    CallNumber : 1..99
    DialNumber : 1..9
    WhoPhoned : STRING[10]
    EntryPrio : TEntryPrio
    DialResult : TDialResult
    ExitPrio : TExitPrio
    DialLineNr : 1..15
    AppointType : TAppointType, EMPTY
    ExitTime : TIMETYPE
    SecondsDial : INTEGER[8]
    SecondsInt : INTEGER[8]
```

The use of the history file as a transaction file is shown in *HISTORY2.MAN*.

```
PROCESS HistoryPlus

  SETTINGS
    DATEFORMAT = YYMMDD
    PROGRESS=YES

  USES
    HISTORY 'history'
    ComTel 'ComTel'
```

```
INPUTFILE
  histin : history ('comtel.~th', ASCII)
  SETTINGS
    SEPARATOR = ','

UPDATEFILE
  ComData : comtel ('comtel', BLAISE)
  SETTINGS
    ACCESS=SHARED
    ONLOCK=CONTINUE
    CONNECT=NO

MANIPULATE
  {Need DLL copy instruction here.}
  WHILE NOT (HistIn.EOF) DO
    HistIn.READNEXT
    IF HistIn.ExitPrio IN [NoAnswer] THEN
      ComData.GET(HistIn.ThePrimKey)
      IF ComData.RESULTOK THEN
        ComData.DAYBATCH_ADD  {This line continues.}
       (STRTOTIME('07:40'),STRTOTIME('15:00'),HARD,'')
      ENDIF
    ENDIF
  ENDWHILE
```

This simple example sets all *no answer* outcomes for hard appointments between 7:40 and 15:00. It is possible to have a much more elaborate rescheduling scheme than the one shown here.

History file copy with DLL

If the history file is going to be used as a transaction file in a Maniplus setup during calling it is necessary to copy it to a different file name with a DLL program which can be obtained by special request. The DLL locks the history file while the copy is being made so that no information is lost during the copy. If the history file were to be copied to a different name without the locking, it is possible that some history data would not be written to it during the copy. The DLL copy program protects from this.

Modifying a record in the day batch

To modify a record in the daybatch, add a record with an existing ID and it will overlay the original record in the day batch having the same ID. This is shown above.

DAYBATCH_ADD parameters

The key word *daybatch_add* takes four parameters that determine how the form is scheduled in the call management system. They are starting time, ending time, priority, and a string that can name a target group of interviewers or an interviewer. The parameters can be hard-coded as in *HISTORY2.MAN* or they can be *AUXFIELDS*

```
ComData.DAYBATCH_ADD  {This line continues.}
    (CalcStart, CalcEnd, CalcPriority, CalcTarget)
```

DAYBATCH_DEL technical note

The *daybatch_del* command does not physically delete a form from the day batch. Rather, it gives it the 'no need today' status. This may be considered to be a logical deletion.

DAYBATCH_ADD technical notes

When *daybatch_add* is invoked, a form from the Blaise data file is added to the day batch. If there are less than 2000 forms in the day batch no records are physically deleted from the day batch. If there are 2000 forms in the day batch, records with the 'no need today' status are physically deleted. If there are no records with the 'no need today' status then active records are physically deleted from the daybatch.

Index

C

D

L

M