Blaise Developer's Guide

Blaise 4.5 *A Survey Processing System* *Copyright © 2002 by Statistics Netherlands*
Blaise is a registered trademark of Statistics Netherlands.

Blaise® Developer's Guide
Statistics Netherlands
Methods and Informatics Department
Heerlen
Copyright © 2002 by Statistics Netherlands

ISBN 90 3572995 1

Acknowledgments

The Blaise System is developed by Statistics Netherlands. Special thanks for the editing and revision of this manual go to Westat, the licensor and distributor of Blaise Software in North America.

Table of Contents

1 Introduction 1
1.1 Additional Capabilities 3
1.2 Advanced Blaise products 3
1.3 Blaise Documentation 4
1.4 Blaise Examples 5
1.5 Conventions Used in This Guide 5
2 Blaise Control Centre 7
2.1 Opening the Control Centre 7
2.1.1 File types in the Control Centre 9
2.1.2 Blaise text editor 11
2.2 Control Centre Functions 18
2.2.1 Prepare command 18
2.2.2 Build Command 20
2.2.3 Run command 21
2.2.4 Setting run parameters 22
2.2.5 Setting General Environment Parameters 23
2.2.6 Projects 25
2.2.7 Data model properties 31
2.2.8 Data file management 31
2.2.9 Configuring tools 32
2.2.10 Manipula Wizard 35
2.2.11 Monitor utility 38
2.2.12 Hospital utility 40
2.2.13 Command line prepare utility 42
2.2.14 A remark on OleDB 42
2.3 Structure Browser 43
2.3.1 Viewing the structure 43
2.3.2 Structure Browser options 47
2.3.3 Viewing data model statements 48
2.4 Database Browser 50
2.4.1 Viewing the data 51
2.4.2 Database Browser options 56
2.5 Help 57
2.5.1 What's New 57
2.5.2 Enter registration 57
3 Data Model Basics 59

3.1 Blaise Language Overview....59
3.2 Fields....69
3.2.1 Field types....74
3.2.2 TYPE section....88
3.2.3 Answer attributes....93
3.2.4 Enhancing texts....97
3.3 Auxiliary Fields (Auxfields)....102
3.4 Local Variables (Locals)....104
3.5 Summary of Fields, Auxfields, and Locals....106
3.6 Rules....107
3.6.1 Route instructions....107
3.6.2 Route field methods....107
3.6.3 Conditional rules....113
3.6.4 Edit checks....120
3.6.5 Computations....132
3.6.6 Looping through rules....134
3.6.7 Layout elements in the RULES section....137
3.6.8 Rules or no rules....138
3.6.9 Empty RULES section....138
3.7 SETTINGS Section....139
3.7.1 Key fields....139
3.7.2 Languages....142
3.7.3 TLANGUAGE, a provided language type....144
3.8 Functions....146
3.9 Data File Compatibility....147
3.9.1 Causes of data file incompatibility....148
3.9.2 Production or development....149
3.10 Good Programming Practices....150
3.11 Example Data Models....151
4 Blocks and Tables....153
4.1 Blocks....155
4.1.1 Blocks as types, repeating code....157
4.1.2 Block-level text....160
4.1.3 Passing information to a block by direct reference....161
4.1.4 Two or more separate blocks....162
4.1.5 Nested blocks....164
4.2 Parameters....167
4.2.1 Parameter example....167
4.2.2 Parameter details....169
4.3 Included Files....173
4.3.1 Format of the INCLUDE command....173

4.3.2 FIELDS and RULES sections in included files 175
4.3.3 File name extensions 175
4.4 Tables 175
4.4.1 Extremely large tables 177
4.4.2 Different kinds of tables 180
4.4.3 Protecting blocks and tables from further change 181
4.5 Mini-data Models 182
4.6 Block Computations 182
4.7 Array Methods 183
4.8 Helpful Administrative and Survey Management Blocks 185
4.8.1 Nonresponse block 188
4.8.2 Appointment block 190
4.9 Parallel Blocks 191
4.9.1 Blocks chosen by menu 193
4.10 Hierarchical Data Models 195
4.10.1 Connecting arrayed blocks 197
4.11 Selective Checking Mechanism and Instrument Performance 198
4.11.1 Performance and parameters 198
4.11.2 Other performance gains 200
4.12 Good Programming Practices 201
4.13 Example Data Models 202
5 Special Topics 205
5.1 Hierarchical Coding 205
5.1.1 Classification type 206
5.1.2 Building the classification type 208
5.1.3 Classify method for coding a field 210
5.1.4 Using the code later in the data model 212
5.2 Retrieving Information from External Files 214
5.2.1 External file requirements 214
5.2.2 The external data model and data file 215
5.2.3 Referring to the external data model and data file 217
5.2.4 Accessing the external data with file methods 220
5.3 Lookups 224
5.3.1 External lookup file 225
5.3.2 Keys in the external lookup file 226
5.3.3 Declaring the external file lookup from the main data model 229
5.3.4 Accessing related data in the lookup record 230
5.3.5 Giving the lookup a starting value 231
5.3.6 Using hierarchical coding and lookup together 232
5.4 Blaise Procedures 233

5.5 Dynamic Link Libraries....236
5.5.1 Two types of alien DLL reference....237
5.5.2 Delphi™ DLLs and other DLLs....237
5.5.3 Delphi™ DLL procedure called by a Blaise DEP alien procedure....238
5.5.4 Delphi™ DLL procedure called by a Blaise DEP alien router....238
5.6 Audit Trail....238
5.6.1 Audit trail DLLs....239
5.6.2 Invoking the audit trail DLL....245
5.6.3 Contents of the audit trail file....246
5.6.4 Miscellaneous audit trail information....247
5.7 Multimedia Language....248
5.7.1 Implementing the multimedia capability....248
5.7.2 Declaring the multimedia language....249
5.7.3 Multimedia key words in the multimedia language....249
5.7.4 Multimedia settings in the mode library file....251
5.7.5 Other multimedia considerations....252
5.8 Question-by-Question Help....254
5.8.1 Using WinHelp....255
5.8.2 Create a WinHelp file....258
5.8.3 Blaise help language....258
5.9 LAYOUT Section....259
5.9.1 Implementing LAYOUT sections....263
5.9.2 Location key words....264
5.9.3 Layout style key words....264
5.9.4 Location and layout key words used together....265
5.10 Example Data Models....265
6 Data Entry Program....267
6.1 Overview of Screen Design in Blaise®....267
6.2 DEP Window Components....268
6.2.1 FormPane....269
6.2.2 Grid....270
6.2.3 FieldPane....271
6.2.4 InfoPane....272
6.2.5 Menu, Speedbar, and Status bar....273
6.3 Modes of Behaviour....274
6.3.1 Routing....274
6.3.2 Checking....275
6.3.3 Error reporting....275
6.3.4 Combining the behaviour modes....275
6.4 DEP Customisation Files....277
6.5 Mode Library File....278

6.5.1 Using the Mode Library Editor....281
6.5.2 Mode library file: Style settings....284
6.5.3 Mode library file: Toggles....292
6.5.4 Mode library file: Layout—Grids, FieldPanes, InfoPanes....300
6.5.5 Viewing pages in the Mode Library Editor....313
6.5.6 Common screen layout tasks....317
6.5.7 Applying a mode library file....323
6.5.8 Detaching/Attaching a mode library file from a data model....324
6.6 Data model properties....324
6.6.1 Set properties for system and user-defined types of the data model....325
6.6.2 Specify text for parallel blocks....327
6.6.3 Languages properties....329
6.6.4 Status bar properties....330
6.7 DEP Configuration File....331
6.7.1 Using the DEP Configuration Program....332
6.7.2 Editing a DEP configuration file....335
6.7.3 Applying a DEP configuration file....335
6.8 Menu File and the DEP Menu Manager....336
6.8.1 Using the DEP Menu Manager....337
6.8.2 Editing and adding menu items....338
6.8.3 Editing and adding speed buttons....343
6.8.4 Applying a menu file....345
6.9 Screen Layout Considerations....345
6.9.1 Data density in the page....345
6.9.2 Font sizes....345
6.9.3 New pages created for new Grids....346
6.9.4 Screen resolution....346
6.9.5 Summary of screen layout factors....346
6.10 Using the DEP....350
6.10.1 Invoking a behaviour mode: interviewing or data editing....351
6.10.2 Entering responses....353
6.10.3 Navigating between forms....357
6.10.4 Errors....360
6.10.5 Languages....362
6.10.6 Multimedia....363
6.10.7 Watch window....363
6.11 Running the DEP Outside the Control Centre....364
7 Basic Manipula....367
7.1 Things You Can Do With Manipula....368
7.2 Starting Manipula....369
7.2.1 Creating a Manipula setup....369

7.2.2 Preparing a Manipula setup 372
7.2.3 Running a Manipula setup 373
7.2.4 Manipula Run parameters 373
7.3 Inspecting Input and Output Data 375
7.4 Basic Operation of Manipula 377
7.5 File Formats Supported by Manipula 378
7.6 Outline of a Basic Manipula Setup 379
7.6.1 USES section 379
7.6.2 INPUTFILE section 381
7.6.3 OUTPUTFILE section 382
7.6.4 MANIPULATE section 382
7.6.5 Other file sections 384
7.7 Basic Examples 384
7.7.1 Extending a Blaise data file 385
7.7.2 Initialising a Blaise data file 385
7.7.3 Exporting a Blaise data file to ASCII 386
7.8 Extending a Manipula Setup 387
7.8.1 AUXFIELDS section 388
7.8.2 SORT section 389
7.8.3 PRINT section 390
7.8.4 SETTINGS section 391
7.9 Running Manipula as a Separate Program 397
7.10 Example Manipula Setups 398
8 Advanced Manipula 399
8.1 More Sections in Manipula 399
8.1.1 PROLOGUE section 399
8.1.2 UPDATEFILE section 400
8.1.3 TEMPORARYFILE section 400
8.2 More About Files 401
8.2.1 Linking files and the LINKFIELDS subsection 401
8.2.2 Day file 402
8.2.3 Message file 402
8.2.4 Customised information files 403
8.2.5 File methods WRITE, KEEP, WRITEALL, and KEEPALL 403
8.3 Example File Structures 404
8.3.1 Address and roster information in one file 404
8.3.2 Address and roster information in separate files 405
8.4 More About MANIPULATE 407
8.4.1 Checking rules 407
8.4.2 Form correctness status 408
8.4.3 Block history 409

8.4.4 Counting forms 409
8.4.5 AUTOREAD = NO 410
8.4.6 Procedures 411
8.4.7 Block computations 412
8.4.8 Functions 413
8.4.9 Exits from loops..413
8.4.10 Stopping Manipula..414
8.4.11 Debugging Manipula setups 415
8.5 Manipula and Its Environment 416
8.5.1 Command line parameter strings 416
8.5.2 Environment variables...417
8.5.3 Local area network (LAN) issues 418
8.6 Reformatting Files 420
8.6.1 One physical record to many 420
8.6.2 Many physical records to one 423
8.7 Importing Blocks of Data Into Blaise 428
8.7.1 Address and roster information in one file 428
8.7.2 Address and roster information in separate files 430
8.7.3 Two-stage ASCII read-in with UPDATEFILE 430
8.7.4 Reading in two ASCII files at the same time...434
8.8 Exporting Blocks of Data from Blaise...437
8.8.1 ASCIIRelational file types...437
8.8.2 EMBEDDED and ordinary blocks 439
8.8.3 Exporting one or a few blocks of data 442
8.9 Miscellaneous Uses of Manipula...444
8.9.1 Making a test data set 444
8.9.2 Creating a library file for classify...445
8.10 Performance Issues...445
8.10.1 Improving performance with Manipula features..445
8.10.2 Skipping to a secondary key value..446
8.10.3 Data sharing 448
8.10.4 Filters 448
8.10.5 TEMPORARYFILE 449
8.10.6 Block computations 449
8.10.7 CONNECT = NO...450
8.10.8 AUTOCOPY = NO...450
8.11 Example Manipula Setups...450
9 Cameleon 453
9.1 Cameleon and Metadata 453
9.2 Example Data Model 454
9.3 Cameleon Translators Supplied with Blaise..456

9.4 How to Start Cameleon....457
9.4.1 Running Cameleon....458
9.4.2 Setting Cameleon run parameters....459
9.5 Cameleon Output Samples....460
9.5.1 Output from spss.cif....460
9.5.2 Output from sas.cif....463
9.6 Programming in Cameleon....465
9.6.1 Basic Cameleon programming concepts....466
9.6.2 Example program cameltst.cif....466
9.6.3 Example program param.cif....468
9.6.4 Example program wesvar.cif....468
9.6.5 Analysing the wesvar.cif translator....472
9.6.6 Using metadata loops....473
10 CATI Call Management System....477
10.1 Blaise CATI Concepts....478
10.2 CATI Interviewing....483
10.2.1 Make Dial screen....484
10.2.2 Making appointments....485
10.2.3 Using a CATI menu....489
10.3 Developing CATI Data Models....490
10.3.1 INHERIT CATI and TCatiMana....490
10.3.2 Special CATI fields....493
10.3.3 Appointment block....497
10.3.4 Additional blocks....498
10.3.5 Initialise the data file....499
10.4 CATI Specification Program for Study Management....500
10.4.1 Create a specification file....501
10.4.2 Survey days....503
10.4.3 Crew parameters....505
10.4.4 General parameters....507
10.4.5 Dial menu....511
10.4.6 Field selection....513
10.4.7 Interviewers and Groups....517
10.4.8 Time zones....522
10.4.9 Time slices....523
10.4.10 Quota control....525
10.4.11 Parallel blocks....528
10.4.12 Daybatch select....530
10.4.13 Daybatch sort....533
10.5 CATI Management Program for the Supervisor....533
10.5.1 Create daybatch....535
10.5.2 Summary....542

10.5.3 Forms 543
10.5.4 View active interviewers and groups 546
10.5.5 Set environment options 546
10.5.6 View history and log files 547
10.5.7 Configure the Tools menu 550
10.5.8 Running the CATI Management Program outside the Control Centre 550
10.6 Example: A Simple CATI Survey 551
10.6.1 Step 1: CATI data model 551
10.6.2 Step 2: Initialising the data file 553
10.6.3 Step 3: Survey specification 555
10.6.4 Step 4: Survey management 555
10.6.5 Step 5: Interviewing 556
10.7 CATI/CAPI Compatibility 558
10.8 Other Considerations 559
11 CATI Technical Details 563
11.1 Rules for Inclusion in the Daybatch 563
11.2 Call Scheduler 565
11.2.1 Selecting forms 566
11.2.2 Routing back forms 567
11.2.3 Assigning priorities, starting times, and ending times 568
11.2.4 Activating a form with medium or higher priority 571
11.3 Treatments 573
11.3.1 Treatment of dials 575
11.3.2 Exceptions to general treatment rules 575
11.4 Files Needed for CATI 576
11.5 History File 577
11.6 Glossary 580
Appendix A: Command Line Parameters 585
Command line prepare utility (B4CPars.exe) 585
Cameleon (cameleon.exe) 586
CATI Emulator (btemula.exe) 586
CATI Management Program (btmana.exe) 586
CATI Specification Program (btspec.exe) 587
Control Centre (blaise.exe) 587
Data Entry Program (dep.exe) 587
Hospital (hospital.exe) 589
Manipula/Maniplus (manipula.exe) 589
Blaise Command Line Option Files 590
Appendix B: Files in Blaise 595

Instrument Files 595
Blaise Data Files 596
External Data Files 597
DEP Customisation Files 597
Data Entry Program Files for Stand-alone or Remote Operation 598
Manipula/Maniplus Files for Stand-alone or Remote Operation 598
Files for Distribution for an Application 598
Source Code Files 599
Folder Structures 600
CATI Call Management System Files 601
Index 603

1 Introduction

Blaise® is a powerful and flexible system used for computer-assisted survey processing. Blaise® can perform Computer-Assisted Telephone Interviewing (CATI), Computer-Assisted Personal Interviewing (CAPI), Computer-Assisted Self-Interviewing (CASI), interactive editing, high-speed data entry, and data manipulation and has full survey management capabilities. With Blaise® you can perform various activities in an easy and user-friendly way.

Blaise® is used worldwide by many types of survey organisations, including government, university, and private research companies. These organisations conduct a wide variety of surveys such as labour force surveys, consumer price surveys, multilevel rostering household surveys, panel surveys, business and economic surveys, institutional surveys, health surveys, energy surveys, environment surveys, agricultural surveys, and programs of related surveys.

Blaise features

Blaise provides a multitude of options and features for the survey developer:

- Virtually unlimited capacity for extremely large numbers of questions, edits, and hierarchies
- Constant enforcement of all appropriate routes and edits without slowing down during long interviews
- Concurrent interviewing of two or more respondents
- Hierarchical, alphabetical, and trigram coding schemes which can be used together
- Lookups of information held in external files
- Metadata management and manipulation
- Data manipulation, re-coding, exporting, and importing
- Language switching during interviewing programmed to happen automatically, through menus, or with a keystroke
- Question-by-question interview aids within Blaise itself or through WinHelp
- Multimedia capability for graphics, video, and audio

- Mouse, pen, and touch-screen support
- Survey management and reports
- Sophisticated CATI call scheduling, including time zone adjustments, interviewer assignments, and time slices to target respondents
- Full control of fonts and font sizes for questions, response text, and entry cells
- Customisable user interfaces which can be modified for your organisation
- An audit trail that can be customised for your own needs

Beneficial for all

As a development system, Blaise is suitable for both the individual and the large survey organisation. A control centre integrates many tools that help the developer produce and test instruments. Blaise can be used for surveys in multiple modes such as CATI and CAPI combined collections.

For interviewers, data entry personnel, and data editors, the interfaces are powerful and elegant and have proven to be very efficient and popular. For methodologists, Blaise allows data to be gathered correctly by using edits during the interview.

For high-level managers, Blaise can increase productivity. During instrument development, one system specification handles many tasks. During survey production, data collection, coding, entry, and editing are all combined into one or a few steps.

For systems managers, Blaise is a powerful, generalised system that can be customised to the organisation’s needs, avoiding the need to develop expensive in-house systems.

Easy to use

Subject matter specialists, statisticians, and programmers can become adept at authoring Blaise instruments. The modular and reusable structure of the language allows many surveys to use the same blocks of code with little or no modification. This results in faster, surer development and better comparability between surveys. The multi-mode nature of Blaise encourages (and can enforce!) consistent specifications and conventions between multiple modes of use.

1.1 Additional Capabilities

Use of DLLs:

There is little the Blaise language cannot do, but if you have a special need, you can use Dynamic Link Libraries (DLLs) to extend the system. You can use DLLs to display graphics, read data from a serial port, invoke specialised coding software, or perform a complex calculation already programmed elsewhere.

Connecting to other processes:

Within a Blaise survey application one can interact with other processes, running executables, DLLs, and ActiveX® components from the survey menu.

1.2 Advanced Blaise products

The Blaise system also includes separately licensed products that powerfully extend and enhance the core system. These advanced products are documented in separate publications.

Maniplus

Maniplus is an interactive menuing and execution system that can build survey management systems around instruments to organise the users' many tasks. With Maniplus you can implement a multi-survey CAPI laptop management system or a data flow and survey management system, each completely customised. Maniplus is value-added.

Blaise Component Pack

The Blaise Component Pack (BCP) consists of a number of COM and ActiveX® components. Using industry-standard development tools such as Microsoft® Visual Basic™, C++®, or Borland® Delphi™, these components allow for instant access to Blaise instrument metadata and data, as well as to relational databases via ActiveX® Data Objects (ADO) /OleDB. The BCP can only be used in combination with the basic system.

1.3 Blaise Documentation

The Blaise documentation consists of:

- *Developer's Guide*
- *Reference Manual* (also available on-line in the Blaise program)
- *Maniplus Manual*

This *Developer's Guide* is organised as a reader, explaining and describing many features of the system. The Reference Manual and the Maniplus Manual contain more detailed technical details.

This reader is arranged in Chapters according to global parts of the Blaise system. The first chapters explain the basics of the Blaise system. The remaining chapters offer you comprehensive descriptions of specific domains within the system.

This *Developer's Guide* includes the following chapters:

- *Chapter 1, Introduction:* Overview of Blaise and introduction to the *Developer's Guide*.
- *Chapter 2, The Blaise Control Centre:* How to use all the features of the Blaise Control Centre efficiently to produce quality data collection and editing instruments.
- *Chapter 3, Data Model Basics:* Covers the basic Blaise language used to construct a data model.
- *Chapter 4, Blocks and Tables:* Introduces blocks, which are an important unit of construction for large applications.
- *Chapter 5, Special Topics:* Covers special topics on data model construction that are needed for some applications but not needed for others. Topics include coding, procedures, Dynamic Link Libraries, external files, layout features, and sophisticated navigation.
- *Chapter 6, The Data Entry Program:* A description of the Data Entry Program (DEP) and how to use and customise it to suit your needs. Topics include the DEP window components, changing modes of behaviour, and customising the interface.

- *Chapter 7, Basic Manipula:* Explanation of the Manipula program for simple data management and survey management activities. Topics include reading data in and out of a Blaise data model, writing reports, and describing data.
- *Chapter 8, Advanced Manipula:* Explanation of how to handle difficult file structures and specialised needs using Manipula.
- *Chapter 9, Cameleon:* How to use the Cameleon metadata utility.
- *Chapter 10, The CATI Call Management System:* How to create a data model for Computer Assisted Telephone Interviewing (CATI) and how to harness the power of the Blaise CATI Call Management System.
- *Chapter 11, CATI Technical Details:* Provides background information and technical details about the CATI Call Management System.
- *Appendix A, Command Line Parameters*
- *Appendix B, Files in Blaise*

1.4 Blaise Examples

In addition to the text of the manual, numerous example data models are included with the system distribution[1], which correspond to examples in the *Developer's Guide*. They use the fictitious National Commuter Survey as a vehicle to illustrate topics from this guide.

Statistics Netherlands wishes to thank the U.S. Department of Transportation’s National Highway Traffic Safety Administration (NHTSA) for providing make, model, and model year data for passenger vehicles. These data were used in several coding examples in the manual. The data were based on vehicles involved in fatal crashes in 1992 as reported in NHTSA’s Fatal Accident Reporting System.

1.5 Conventions Used in This Guide

We have used some standard conventions throughout this manual to make it easy to use and understand.

[1] The example files are available in the subdirectory DOC of the Blaise system directory after a complete installation of the Blaise system. The Blaise system directory is the directory where the file BLAISE.EXE is available.

Menu commands

All menu commands are in the format: *Menu name ➤ Command.* For example, the command *File ➤ Save* means to first select the *File* menu item, and then select the *Save* command within the menu.

Mouse commands

We've used the following terms to indicate mouse commands:

- *Click*: Click the left mouse button once.
- *Double click*: Click the left mouse button twice.
- *Right click*: Click the right mouse button once.
- *Drag and drop:* Use the mouse to select one or more items, hold the left mouse button down, and move the items to another part of the screen or window.
- *Italicised text*: References to windows, dialog boxes, menus, buttons, and speed buttons are italicised in the text.
- ! *Symbol*: Special notes are formatted with this symbol to draw your attention to them. When you see this symbol, you know that the information is a helpful reminder or of special importance
- *Language*: All Blaise language key words are capitalised in the text.

In all cases, the term *folder* is used to refer to a *directory.*

2 Blaise Control Centre

The Blaise® Control Centre is the development shell that contains the Blaise® tools and programs you will work with when developing instruments. From the Control Centre you manage data models, programs, utilities, and configuration files for the instrument.

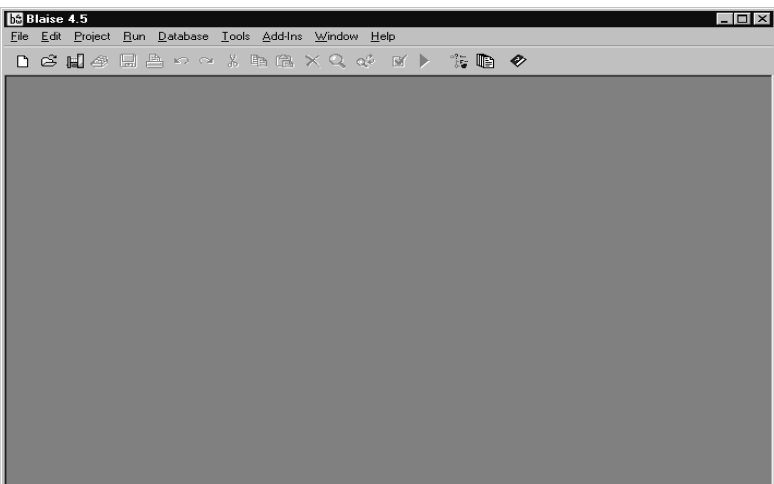
This chapter describes the Control Centre and its various components. Many of the activities initiated from within the Control Centre are also discussed in subsequent chapters.

We assume that you have already installed the Blaise system correctly on your computer. We also assume that you have a good working knowledge of Windows®.

2.1 Opening the Control Centre

Open the Control Centre by clicking its shortcut or by clicking the Windows® *Start* button, selecting the appropriate choices in the *Start* menu, and clicking the Blaise option on the submenu. This will start and open the Blaise Control Centre as shown in the following figure.

Figure 2-1: Blaise Control Centre



When you activate the Control Centre for the first time, you will see an empty desktop with menu options and a Speedbar.

Menus

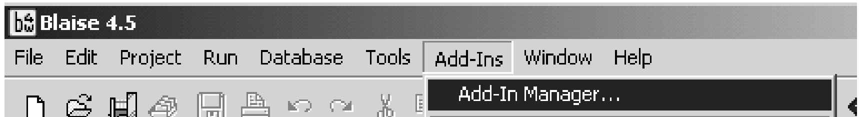
There are several ways to activate menu options: press the Alt key and then the first letter of the menu option; click the menu option with the mouse; or use a shortcut key, which is a special key combination. (Refer to the table of shortcut keys in the *Blaise text editor* section of this chapter.)

When you activate a menu, commands followed by a ➤ symbol have an extended menu box that will appear when you select the command. Commands followed by an ellipsis (…) will display a dialog box for you to provide additional information.

Add-Ins

It is possible to add specialized capabilities to the Control Centre main menu with the Add-Ins menu selection.

Figure 2-2 Control Centre Menu with Add-Ins Menu Selection



Add-Ins launch an ActiveX® control. These ActiveX® controls are custom developed in Visual Basic or other development system. The ActiveX® process may return information to the Control Centre either as a string, or as a file that the Control Centre opens.

Examples of possible Add-Ins include:

- Run an ActiveX wizard that generates some Blaise code and inserts the code at the cursor in the editor, or as a new code window in the editor.
- Run an ActiveX control that accesses a specialized code archive kept in an external database. Select the code desired and it is inserted in the current file.

Details on building an ActiveX control as an Add-In, and using the Add-In Manager to setup the custom menu selection are covered in the Help topic “Add-Ins for the Blaise Control Centre”.

Speedbar

The Control Centre has a Speedbar with speed buttons that allow you to activate some Blaise functions quickly. To see a description of a speed button's function, place the mouse pointer on the speed button but do not click on the button. A small pop-up window with descriptive text called a Tooltip appears.

2.1.1 File types in the Control Centre

There are many types of text files that you can open and work with in the Control Centre. The following table summarises some of the common text files. The file extensions that are listed are recommended but not required. A complete list of all Blaise file types is in Appendix B.

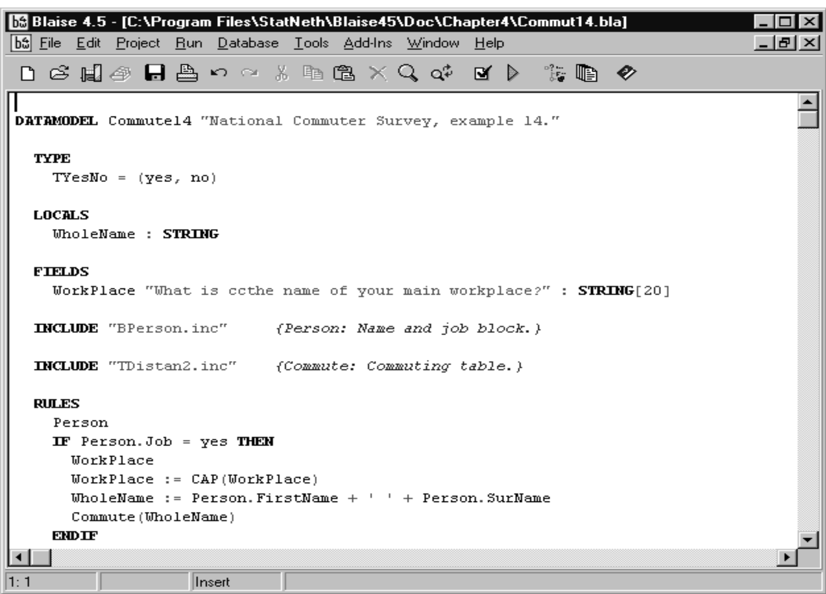
Figure 2-3: Common file types

File Type	Extension	Description
Blaise Data Model	.bla	A file that states the definition and structure of the survey data and their interrelationships. You create instruments from the data model file.
Included File	.inc	A file that contains additional program code, such as sub-parts of a data model or a file manipulation setup.
Library	.lib	A file of type definitions.
Cameleon	.cif	A file that transforms Blaise meta information files into syntax that can be used by statistical or database programs.
Manipula or Maniplus	.man	A file that contains program code to move or manipulate data using Blaise's file manipulation utility.

Open a file

To open a file, select *File* ➤ *Open* from the menu and select a file. The file appears in the Control Centre. Figure 2-4 shows the Blaise data model file `commut14.bla`, which can be found in `\Doc\Chapter4` of the Blaise system folder.

Figure 2-4: Simple data model



Drag and drop

Blaise follows standard Windows® conventions and therefore supports drag and drop capability. You can drag a text file from the Windows® Explorer or desktop directly into the Blaise Control Centre. The drag and drop method is very useful if you want to open several files in the Control Centre at one time. You can select files in the Windows® Explorer and then drag and drop them into the Control Centre in one step.

Open a file from within a file

You can open a text file from within a Blaise file. For example, in a data model you might reference another file that contains information needed to run your model. You can open this file simply by placing your cursor on the file name and pressing Ctrl-Enter. In the above example, the line `INCLUDE BPerson.inc` references such a file. If you placed the cursor directly on the text `BPerson.inc` in the file, and then pressed Ctrl-Enter, the `BPerson.inc` file would open.

View file history list

You can view a list of your most recently opened files. Select *File* ➤ *Reopen* and the *Reopen file* dialog box appears. Scroll through the list, select the appropriate file, and click the *OK* button.

2.1.2 Blaise text editor

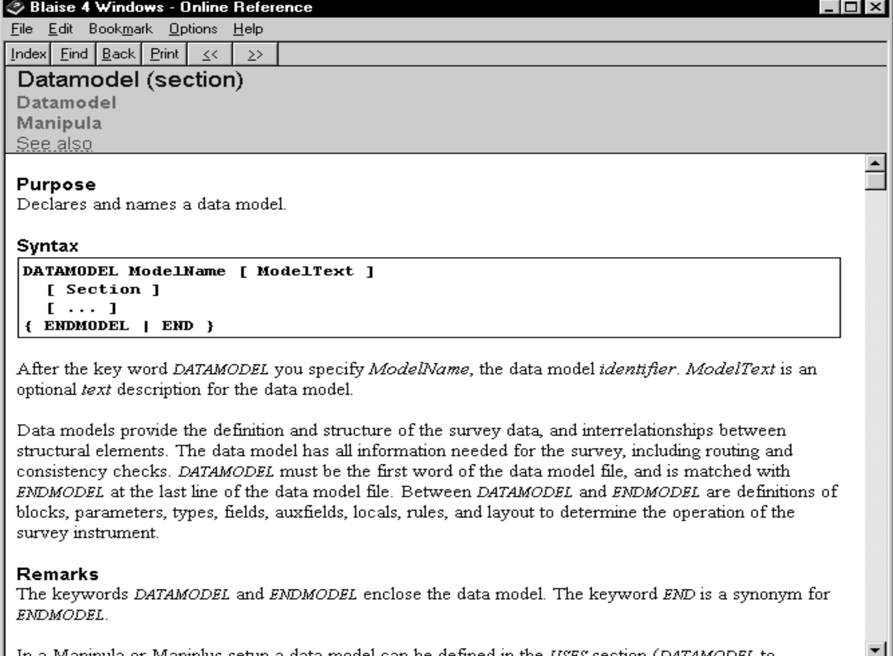
When you open a text file in the Control Centre, you automatically enter the Blaise text editor. We will not discuss the editor in full detail, since it is very similar to many other commonly used editors, but we will mention a few features.

Context-sensitive help

Blaise provides context-sensitive help for key words that appear in files in the text editor. Key words are Blaise language words. (The Blaise language is discussed in detail in Chapters 3, 4, and 5.) With the cursor on a key word, press F1. The Blaise Help window appears, showing the topic related to the key word. If you press F1 when the cursor is not on a key word, the Editor help topic appears.

For example, if you placed the cursor on the key word DATAMODEL, the following Help window would appear.

Figure 2-5: Help window for key word



Blaise Help often contains samples of Blaise language syntax. You can easily cut and paste the Help samples into your own data model file.

Shortcut keys

Blaise provides some shortcut keys to make it easier to use the editor. Some of these are standard Windows® shortcuts and others are specific to Blaise. The following table summarises the shortcut keys. You can select a block of text as in any other Windows® editor or word processor by using the mouse or pressing the Shift and down arrow keys simultaneously.

Figure 2-6: Editing shortcuts

Editing Shortcuts	Description
Ctrl-A	Select all the text in the file
Ctrl-C	Copy a block of text
Ctrl-K-B	Mark the start of a block for indenting or outdenting text (see Tab, Shift-Tab)
Ctrl-K-K	Mark the end of the block for indenting or outdenting text (see Tab, Shift-Tab)
Ctrl-R	Replace text
Ctrl-V	Paste text
Ctrl-X	Cut a block of text
Ctrl-Z	Undo the last action (performs multiple times)
Shift-Ctrl-Z	Redo the last undo (performs multiple times)
Ctrl-Spacebar	Insert non-printing space
Tab	Move highlighted text over a space
Shift-tab	Move highlighted text back a space
F1	Context sensitive help for word under the cursor
F12	Convert text of entire word at the cursor position from lowercase to uppercase

Figure 2-7: Navigation shortcuts

Navigation shortcuts	
F3	Find next/replace next
F7	Reopen files
F9	Prepare a file
Ctrl-F	Find text
Ctrl-O	Open a file
Ctrl-P	Print a file
Ctrl-S	Save a file
Ctrl-F4	Close the editor window and all files displayed on different tabs
Shift-F4	Close the file of the currently active tab
Ctrl-F6	Go to another window
Ctrl-F9	Run a file
Ctrl-Enter on a file name	Open that file
Home	Go to the beginning of the line
End	Go to the end of the line
Ctrl-Home	Go to the beginning of the file
Ctrl-End	Go to the end of the file
Ctrl-Tab	Go to next tab within one edit window
Ctrl-Shift-Tab	Go to previous tab within one edit window

Status bar

There is a status bar at the bottom of the editor window. The status bar displays the line number and column number at which the cursor is positioned. When the cursor is on the very first character of the very first line of a file, this number is *1:1.* As you move the cursor around in the file, this number changes to reflect the cursor's position.

Go to a line

You can go to a specific line number of your file. Select *Edit* ➤ *Go to line* from the menu. Type a line number in the dialog box and click the *OK* button.

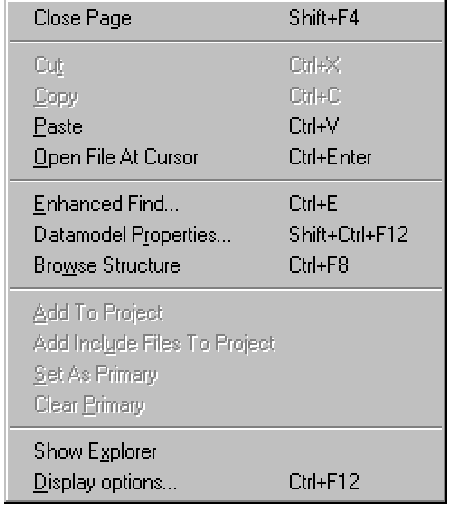
Indenting/outdenting block

The editor supports indenting and outdenting a selected block by using the Tab/Shift-Tab key. One may select a block using the mouse or the keyboard. With the keyboard, use Ctrl-K-B to mark the start of the block and Ctrl-K-K to mark the end of the block. See Editor keys for a review of all available special editor keys.

Editor pop-up menu

A very flexible way to handle many functions and options with the Blaise editor is to use the pop-up menu. Right-click when the cursor is over the editor and the menu appears. Now one can close the current page, open a file named at the cursor's current position, and use the following additional functions:

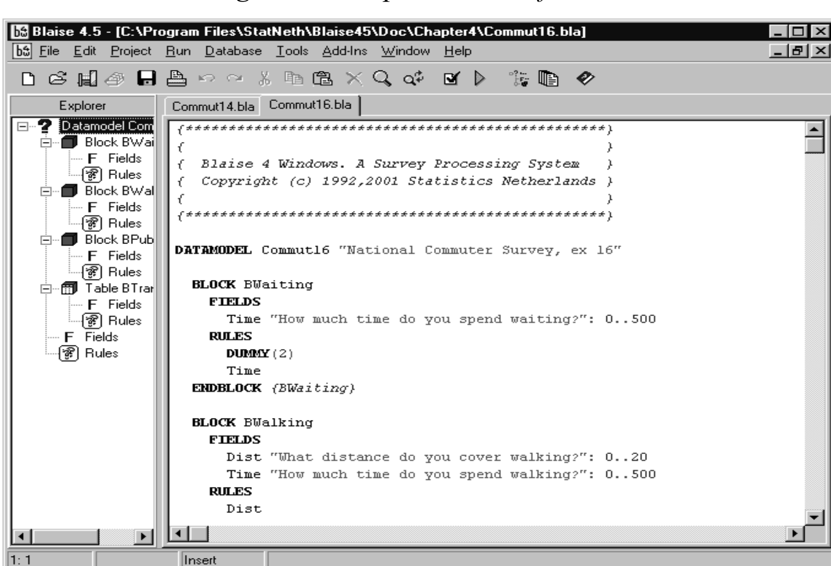
Figure 2-8: Editor pop-up menu



- *Enhanced find.* The enhanced find allows you to search for a string across all include files used by your data model or Manipula setup. You can limit the search to comments only or search all text. Find locates all lines of code containing the string and displays them in the result list window. Scroll through the result list to locate the instance for which you are looking. Pressing the space bar while you have the focus on the instance in the result list, or double clicking on the item with the mouse, highlights the particular instance in the editor. If necessary, it will open the file and transfer the focus to the file containing the highlighted instance. For details open the Enhanced Find form and press F1.

- *Show Explorer*. A special view on your source code in an edit file is also available in the editor. You can use this view to navigate your source code, even if the source code is not syntactically correct yet. There are two ways to enable this view: either right-click on the Editor window and select *Show Explorer* from the pop-up menu or select this option from general environment options. Now in the left-side panel is a tree structure showing the blocks in the data model. Clicking on *Fields* or *Rules* will cause Blaise to move the cursor to the selected Fields or Rules sections in the editor panel. Double clicking on a particular file will open that file in the current window.

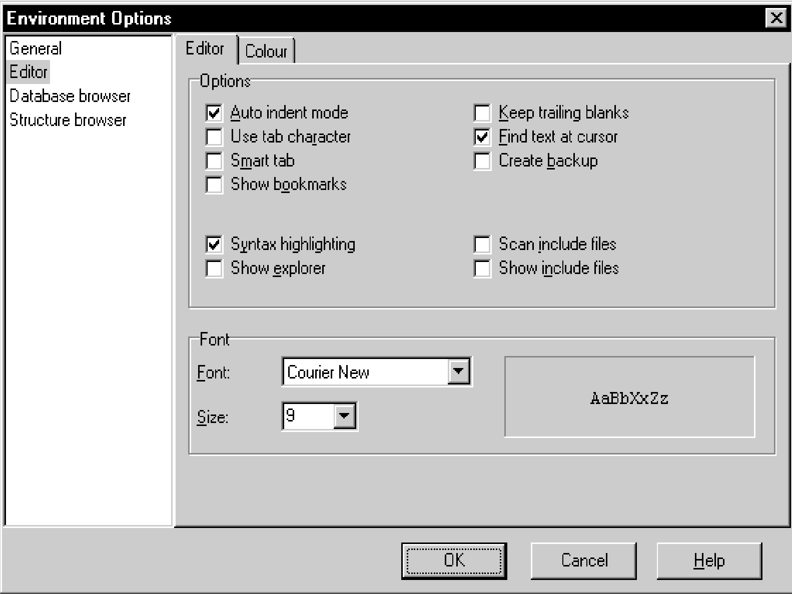
Figure 2-9: Explorer view of source



Editor options

You can set options for the Blaise text editor. To set options for the currently active window only, right-click on the Editor window and select *Display options* from the pop-up menu. To set options for the current window and all subsequently opened files, select *Tools* ➤ *Environment Options* from the menu and select the *Editor* tab.

Figure 2-10: Environment Options

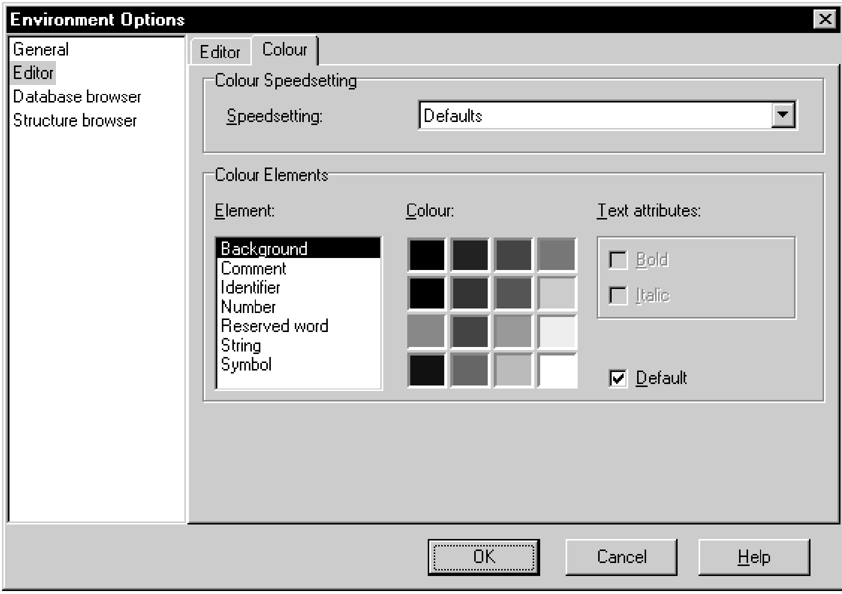


Select options as described below.

- *Auto indent mode.* Pressing the Enter key will move the cursor to a new line under the first non-blank character in the preceding line. This option is useful to keep your source code readable.
- *Use tab character.* Blaise will recognise a tab as a character and not just as white space.
- *Smart tab.* Tab stops will automatically follow the spacing of the previous line in the data model, indenting to the next full word. This makes indenting in your data model easier.
- *Show bookmarks.* Bookmarks that have been set in the margin of the editor windows will be shown. If set, the margin will be wider.
- *Keep trailing blanks.* Blaise will recognise spaces at the end of a line. If this option is not checked, any spaces inserted at the end of a line will be ignored.
- *Find text at cursor.* Blaise will automatically insert the word at which the cursor is placed when you invoke *Find.*
- *Create backup.* Blaise will create a backup file each time you save a file. The backup file will have a `.bak` extension.

- The *Font* and *Size* boxes change the display font of the text editor. Only non-proportional fonts are available in the Blaise text editor.
- *Syntax highlighting*. Syntax highlighting changes the colours and attributes of your text in the Editor, making it easier to identify quickly parts of your code. You can set the colours and attributes for the following different elements of your text: Background, Comment, Identifier, Number, Reserved word, String, and Symbol. The system comes with a predefined list of tokens that will be treated by the editor as reserved words. It is possible to modify this list by adding or removing lines from the text file `Blaise.sht`. Syntax highlighting is only available for files with specified extensions. These file extensions can be specified under *Tools ➢ Environment Options ➢ General*.

Figure 2-11: Setting editor colours



Setting editor colours

You can customise the colours used for syntax highlighting by the editor in the Colour tab of the Editor page of Environment Options.

Display several edit files in one edit window

If this option is active when you load a file, a new edit file will be opened in the currently focussed edit window. If the currently active window is not an edit window (it is a structure browser or database browser window), a new edit window will be created when you load a file. You can navigate between the different tabs within one edit window with Ctrl-Tab or Ctrl-Shift-Tab (or by

clicking on the relevant tab). You can close a tab using the close option in the menu or the close page option in the pop-up menu (Shortcut Shift-F4; you can close the complete edit window with Ctrl-F4).

2.2 Control Centre Functions

This section describes some of the Control Centre's major features and functions, including preparing Blaise files, executing programs, organising files into projects, managing data files, configuring tools, setting environment preferences, and creating simple Manipula files.

2.2.1 Prepare command

The Prepare command checks source code files for syntax errors. The process creates prepared files that you will need to continue with various survey development steps. For example, when you successfully prepare a data model, two files are generated: one file with a `.bmi` extension and one file with a `.bdm` extension. These files are called *meta information* files and are used by Blaise when you run the Data Entry Program.

You will use the Prepare command for data models, libraries, Manipula files, Maniplus files, and Cameleon files. The following table shows the files that result from preparing the different file types.

Figure 2-12: Resulting files from the Prepare command

Source Code File Type	Resulting Prepared File
Data model (.bla)	.bdm (screen layouts)
	.bmi (meta information file)
Manipula file (.man)	.msu
Maniplus file (.man)	.msu
Library file (.lib)	.bli
Cameleon file (.cif)	Checks the .cif file; no prepared file created

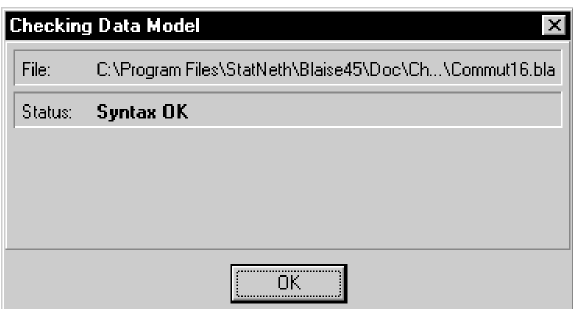
How to prepare

To prepare a file, first open the file. Select *Project* ➤ *Prepare* from the menu, or click the *Prepare* speed button on the Speedbar, or press F9. You do not have to indicate to Blaise the type of file you are preparing; Blaise automatically detects

the file type. If you have created a project and have that project open, Blaise will prepare the primary file for that project. If you have not created a project, or have not set a primary file, Blaise will prepare the file in the active window. Projects are discussed later in this chapter.

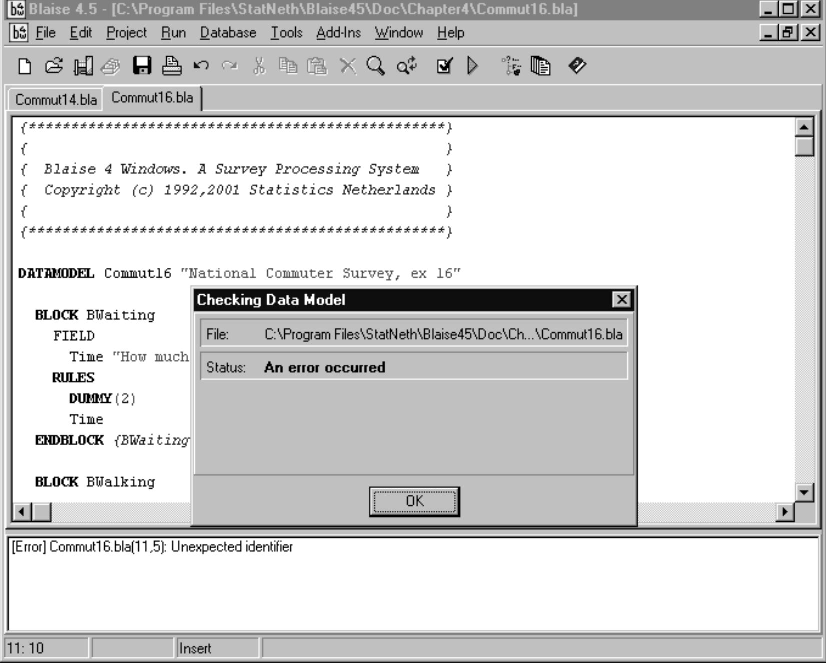
Blaise begins checking the file and a dialog box appears as it does so. If there are no errors, Blaise displays the message *Syntax OK*.

Figure 2-13: Syntax OK message



If there is an error, Blaise displays the message *An error occurred* in the dialog box. Details on the error are shown in the panel at the bottom of the Blaise form. The message includes the name of the file and the position in the data model where the error was found. For example, one of the key Blaise language words is FIELDS. If you have a mistake in your data model and use the word FIELD instead, you will get the following message:

Figure 2-14: Sample error message during prepare



The syntax error will remain visible in the result list window at the bottom of the screen after you have clicked *OK* to close the dialog box. You can navigate to the position of the syntax error in your source code by double clicking on the error you wish to address.

There are other errors that might be more difficult to find, such as a missing end double quote ("), single quote ('), or brace (}). These symbols are very meaningful to the Blaise language. If one is missing in your data model and you receive an error message when you return to your file, the cursor will jump to the spot where Blaise encountered the error, but not necessarily to where the missing element should be. Using syntax highlighting helps identify the source of errors more quickly and easily.

2.2.2 Build Command

Choose *Project* ➢ *Build* if you want to prepare your data model or Manipula setup and all the files on which they depend. The build will prepare all uses data models and all libraries (but only if the source code can be located).

2.2.3 Run command

The Run command allows you to start and run a prepared Blaise file or a program. You can run a data model (which starts the Data Entry Program), Manipula (to manipulate data), Maniplus (for survey management), and Cameleon (to recast Blaise meta information for use by other software packages).

Using Run

To start the Run command, open the file. Then select *Run* ➤ *Run* from the menu, or click the *Run* speed button on the Speedbar, or press Ctrl-F9. You do not need to specify the type of file you are running; Blaise automatically detects the file type.

If you have created a project and have that project open, Blaise will run the primary file for that project. If you have not created a project, or have not set a primary file, Blaise will run the file in the active window.

If, however, the active window contains the Database or Structure Browser, Blaise will start the Data Entry Program (DEP) for the data model that is in the Structure Browser, using either the data file in the Database Browser or the data file indicated in the DEP run parameters. In this case, Blaise will ignore the primary file. The Database and Structure Browsers are discussed later in this chapter.

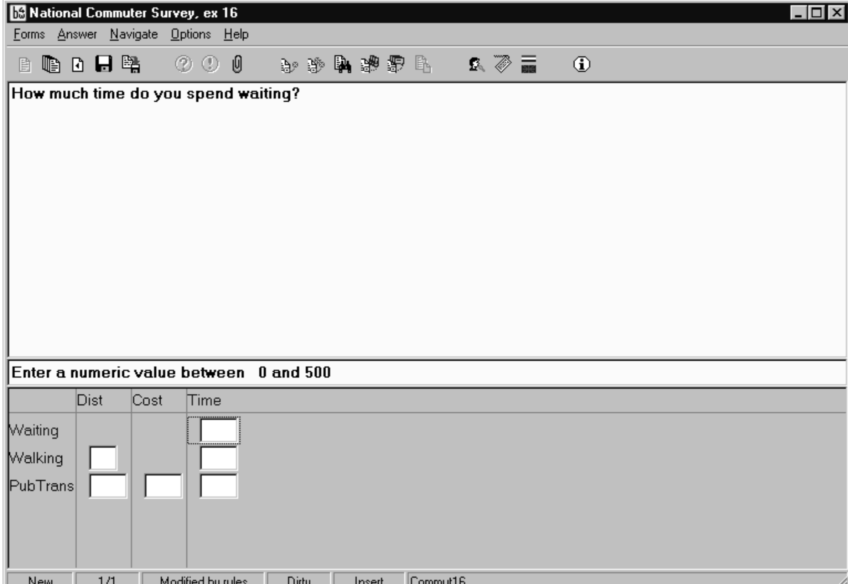
You must first prepare a file before you can run it. If you try to run an unprepared file, or a file that was prepared and then changed, Blaise will prompt you to prepare the file again by displaying the message *Prepared file [filename] on disk is not up-to-date. Prepare again before running?*

An example: Running the Data Entry Program

One use of the Run command is to start the Blaise DEP. First, open and prepare the data model to be used. Select *Run* ➤ *Run* from the menu. The DEP starts up from the Control Centre and appears on your screen.

The following sample shows the DEP when running the prepared `commut14.bla` data model. (`Commut14.bla` can be found in `\Doc\Chapter4` of the Blaise system folder.)

Figure 2-15: Data Entry Program

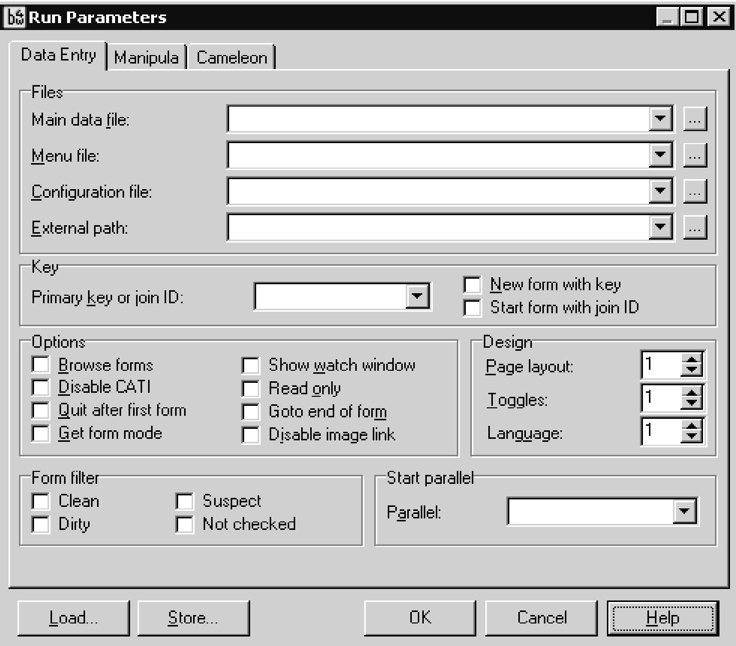


2.2.4 Setting run parameters

When you run the Data Entry Program, you have the option to set certain parameters for the Data Entry Program, Cameleon, and Manipula. The details on setting these parameters are discussed in their respective chapters. We mention them here because you will want to be aware of the options as you create and test your data models.

To set the run parameters, select *Run* ➤ *Parameters* from the menu and the *Run Parameters* dialog box appears. For Data Entry Program parameters, refer to Chapter 6. For Manipula run parameters, refer to Chapter 7. For Cameleon run parameters, refer to Chapter 9.

Figure 2-16: Run Parameters dialog box

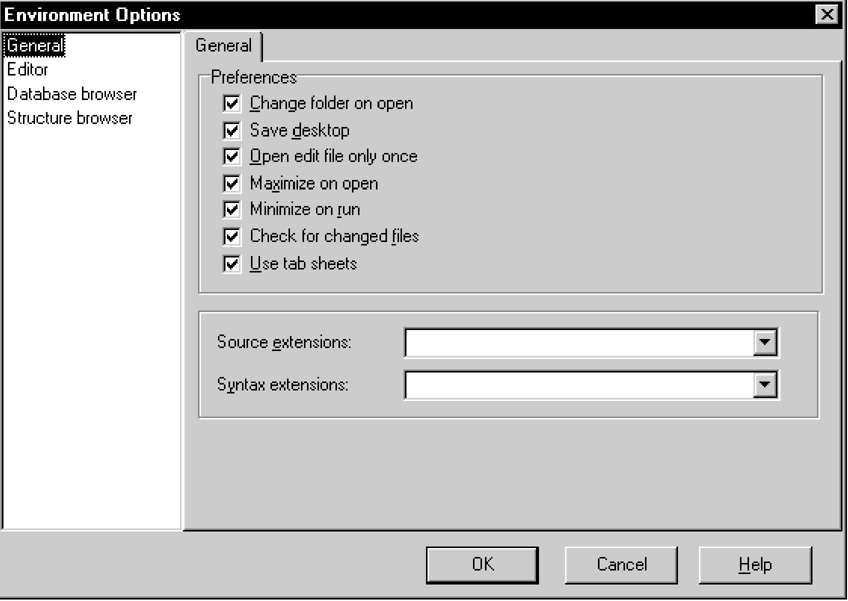


The *Load* button displayed in the bottom of the dialog box allows you to load the command line parameters from a Blaise command line option file. The *Store* button allows you to write the command line parameters to a Blaise command line option file. For more information on Blaise command line option files, see Appendix A.

2.2.5 Setting General Environment Parameters

There are other settings under Environment Options that can be adjusted to customise the Control Centre environment.

Figure 2-17: Environment Options General page



Use the General page of the Environment Options dialog to specify your Control Centre configuration preferences:

- *Change folder on open.* Changes the working folder to the last folder used to open/save a file.
- *Save desktop.* Saves the arrangement of your desktop when you close a project or exit the Control Centre. When you later open the same project, all text files opened when the project was last closed are opened again regardless of whether or not they are used by the project. Database browsers and structure browsers are not reopened.
- *Open edit file only once.* Keeps you from opening the same text file in two separate windows.
- *Minimise on run.* Causes the Control Centre to minimise itself when Run is invoked.
- *Maximise on open.* Causes each newly opened file to be maximised in the Control Centre desktop.
- *Check for changed files.* Causes the Control Centre upon receiving the focus to check if a current edit file has been changed on disk. If yes, you will be prompted to reload that file.

- *Use tab sheets.* Enables using one edit window for multiple edit files. Each edit file can be accessed via a separated tab in the edit window.
- *Source extensions.* Allows you to add to the default file extensions that appear in the File open dialog box when you choose *File* ➤ *Open*.
- *Syntax extensions.* Allows one to specify, by extension, which files will display syntax highlighting information. The default extensions are .BLA, .MAN, .INC, .PRC and .LIB. Also new files (untitled files with no file name) are considered to be files that need to display syntax highlighting information. As soon as you specify new extensions you will also need to include the existing defaults again (if needed).

2.2.6 Projects

When you develop and test an instrument in Blaise, you will probably work with several files at one time. You will also have meta information files and output files that you will need to use.

To help keep track of all your files, it is valuable to use a *project*. A project contains a list of all your files and allows you quick access to them, even if they are in several different folders. You can also specify folders for source files, work files, meta files, and output files for each project. It is an efficient way to keep files organised, and to prevent you from losing files or data during development.

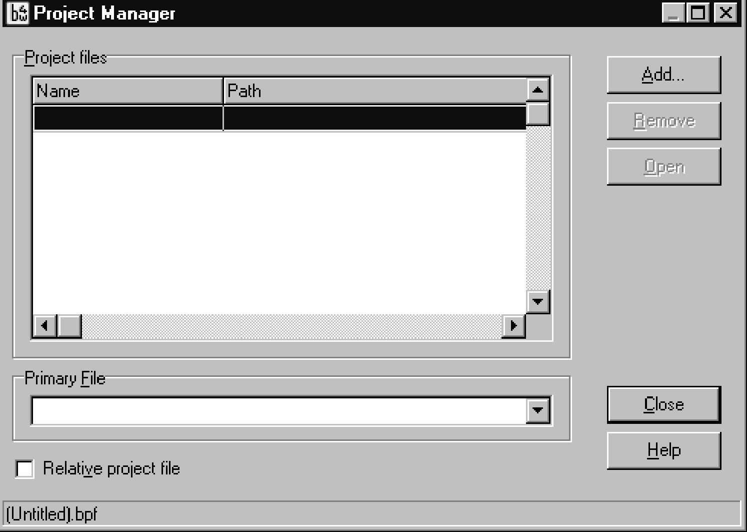
Particularly for larger scale instruments and those with multiple developers, using the Blaise project capability is critical.

Create a project

To create a project, select *File* ➤ *New Project* from the menu. A blank window appears in the Control Centre with the name *Untitled* at the top.

To add files to the project, select *Project* ➤ *Project Manager* from the menu. The *Project Manager* form appears.

Figure 2-18: Project Manager form



Click the *Add* button and select files to add to the project. The file names appear in the Project files list as you add them.

Designate a primary file for the project by clicking the arrow in the *Primary File* box. The primary file is the file that will be run and prepared when those commands are invoked when the project is open. You do not need to have the primary file open to run or prepare it. You can select a primary file from a data model (`.bla`), Manipula (`.man`), or Cameleon (`.cif`) file.

When you check *Relative project file,* all folder and file names stored in the project file will be made relative to the folder where the project file is stored (the so-called project root). This will make it possible to copy a complete project to a different root.

You can also add a file to a project by right clicking on a file in the editor window. A pop-up menu appears.

Figure 2-19: Pop-up menu when right-clicking in editor window

Close Page	Shift+F4
Cut	Ctrl+X
Copy	Ctrl+C
Paste	Ctrl+V
Open File At Cursor	Ctrl+Enter
Enhanced Find...	Ctrl+E
Datamodel Properties...	Shift+Ctrl+F12
Browse Structure	Ctrl+F8
Add To Project	
Add Include Files To Project	
Set As Primary	
Clear Primary	
Show Explorer	
Display options...	Ctrl+F12

From here you can add the file to the project, set the file as the primary file, add the included files to the project, or clear the current primary file.

To save a project, select *File ➤ Save Project As* from the menu. The default file extension is `.bpf`.

Open an existing project

To open an existing project, open the project file. The name of the project and its primary file appear in the title bar of the Control Centre window. The primary file is enclosed by < >. For example, the following sample is the title bar for the project titled `commut.bpf`, and the primary file is `commut14.bla`.

Figure 2-20: The Title Bar of the Control Centre Window

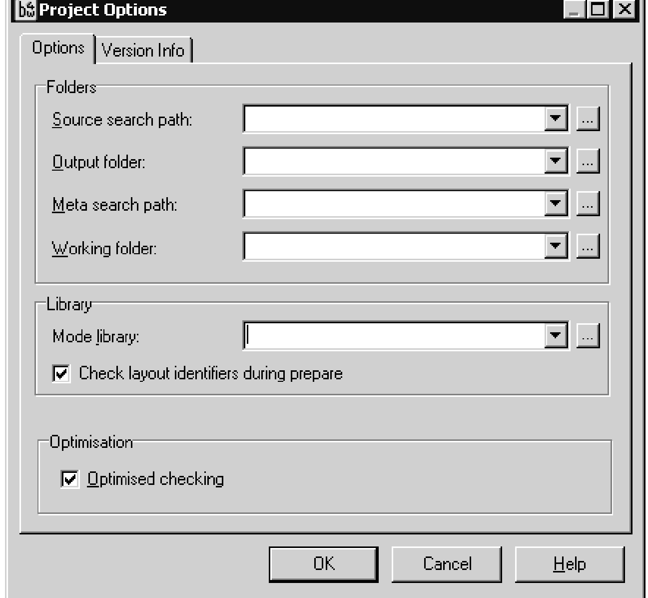
After you have opened the project file, open the Project Manager to access the files.

Project options

You can set various options for a project. You can set project options for a file that is *not* in a project, as well as for an actual project.

When you set project options for a file, such as a `.bla` file, the options will apply to that file when it is open. Using either approach, first open the project, and then select *Project ➤ Options* from the menu. The *Project Options* form appears.

Figure 2-21: Project Options form



In larger Blaise development efforts it is often helpful to place different types of files in separate folders. In the *Folders* section you can specify these folders as described below.

- *Source search path.* Specify a path where Blaise will search for include files for which no path has been specified in the file name. You can specify more than one path. Separate multiple path names with a semicolon (;). Relative and absolute path names are allowed, including path names relative to the logged position in drives other than the current one. Blaise will search the paths until an include file is found.
- *Output folder.* Specify the folder to which the prepared files (`.bmi`, `.bdm`, `.bli`, and `.msu`) will be written. If this box is left blank, they will be written to the folder in which the source file (`.bla`, `.lib`, or `.man`) is located.

- *Meta search path.* Specify the path where Blaise will search for meta files for which no path has been specified.
- *Working folder.* Specify the folder Windows will use as the working folder for other Blaise programs that are run for the project, such as Manipula or the Data Entry Program.

In the *Library* section one identifies a specialised mode library to be used in the project. (See Chapter 6 for more information on the mode library file.)

- *Mode library.* Specify the mode library file to be used when Blaise prepares the data model. If this box is left blank, Blaise will use the mode library file in the working folder; if that isn't present, it will use the file in the system folder.
- *Check layout identifiers during prepare.* Check to display a message if layout identifiers are not found when the data model is prepared. If this is not checked, you will not be notified if Blaise did not find the identifiers.

In the Optimisation section:

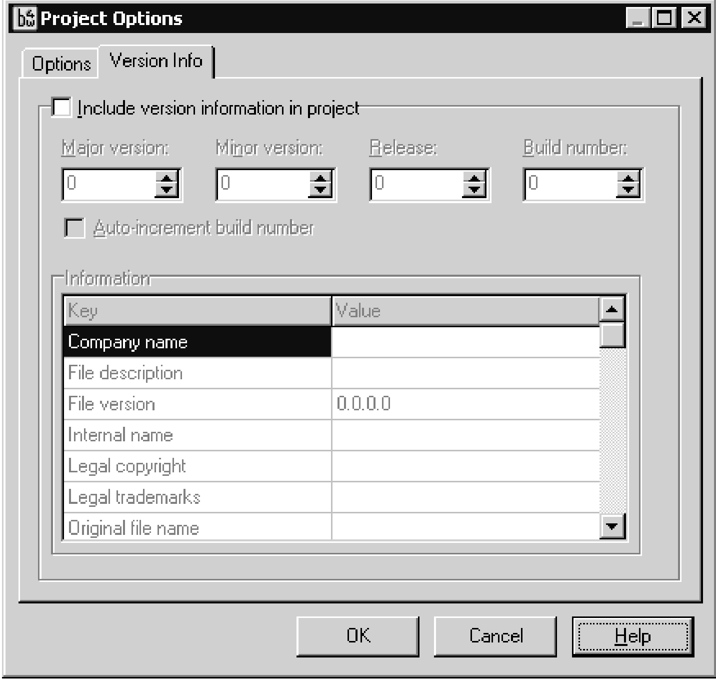
- *Optimised checking.* This option is implemented for backward compatibility only. In general, there is no reason to disable this option. See the on-line help for details. The information on optimised checking is stored in the extended meta file (the .BXI file). This means changing this option requires re-preparing existing models to make use of the optimised checking.

The information on optimised checking is stored in the extended meta file (the .BXI file). In general there is no reason not to check this option.

Project Version Information

Use the *Version Info* tab of the Project|Options dialog to set the version information of your project. This version info will be stored in your prepared data model, your prepared Manipula/Maniplus setup and it will be stored in the Blaise database when created.

Figure 2-22: Project Version Information



Check the *Include version information in project* box to enable this feature. Then version information can be entered and will be included in the prepared data model file or the prepared Manipula setup. Individual information items are:

- *Module Version Number*. Major, Minor, Release, and Build each specify an unsigned integer. The combined string defines a version number for the application.
- *Auto-increment build number.* If enabled, the build number will be incremented each time the Project|Prepare or Project|Build results in a successful preparation.
- *CompanyName.* The company that produced the file.
- *FileDescription*. File description.
- *FileVersion*. File version number.
- *InternalName*. File internal name.
- *LegalCopyright*. File copyright notices.
- *LegalTrademarks*. Trademarks and registered trademarks that apply to file.

- *OriginalFilename*. Original file name, not including path.
- *ProductName*. Name of product that file is distributed with.
- *ProductVersion*. Version of product that file is distributed with.
- *Comments*. Additional information for diagnostic purposes.

You can edit the key value in the Value column.

2.2.7 Data model properties

A large number of default display characteristics of a data model can be customised. This is done using settings in the *Data model Properties* form. The properties are then saved to a file <data model name>`.bxi`.

A full discussion of data model properties requires understanding of important elements in the Blaise language, including types, parallel blocks and languages. These elements are covered in Chapter 3. Then, in section 6.6, we will examine in detail setting data model properties in the Control Centre.

2.2.8 Data file management

Blaise provides a few data file management options from within the Control Centre. You can rename, copy, move, create, or delete a Blaise data file. These options are meant to help you when authoring an instrument, but they do not represent the full spectrum of Blaise data file management. This menu option works only for Blaise data files.

Rename, copy, move, create, or delete

The procedure for each of the file management options is basically the same, with only a few minor variations.

Select *Database* ➢ *Data File Management* from the menu, and then choose the appropriate option. A dialog box appears with a title that corresponds to the option you chose. The following sample shows the dialog box that appears when you select *Copy.*

Figure 2-23: File to Copy dialog box

Select a file and click the *Open* button. The title of the dialog box changes to reflect your action. In the *Copy* example, the title of the dialog box changes to *Copy to*.

If you are renaming a file, type the new file name in the *File name* box. If you are moving or copying a file, you can select a new folder for the file. If you are deleting a file, this step does not occur. Click the *Save* button.

Blaise displays a message to confirm your action. To complete the task, click the *Yes* button; to cancel, click the *No* or *Cancel* button.

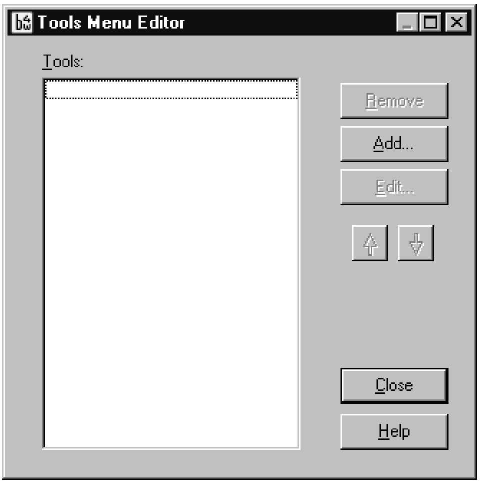
2.2.9 Configuring tools

You can configure the *Tools* menu to run other programs from within the Blaise Control Centre. This is often quicker than leaving the Control Centre, running the program, and then reactivating the Control Centre. When you do this, an option for that program appears in the *Tools* menu. You can configure the *Tools* menu to run Blaise programs as well as non-Blaise programs.

Add a tool

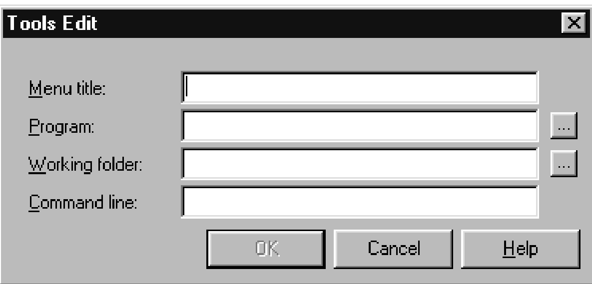
To add a tool, select *Tools* ➤ *Configure Tools* from the menu and the *Tools Menu Editor* dialog box appears.

Figure 2-24: Tools Menu Editor dialog box



If this is the first time you have added tools, the Tools list in the box will be empty. Click the *Add* button and the *Tools Edit* dialog box appears.

Figure 2-25: Tools Edit dialog box



Complete the following items (The only required item is the *Program* box.):

- *Menu title.* Specify the name that you want to be displayed in the menu.
- *Program.* Specify the name of the file that will execute the program you are adding. This file name will most likely have an extension of `.exe`, `.com`, `.bat`, or `.pif`. Use the *Browse* button to select the file, if necessary.
- *Working folder.* Specify the folder Windows will switch to when the program is started.
- *Command line.* Specify applicable command line parameters. Valid parameters are described in Figure 2-26.

Figure 2-26: Command line parameters

Parameter	Description
$ASK( )	Displays a dialog box in which you can specify the command line. In the parentheses, you can specify a value that will be the default value on the edit line of the Ask dialog box.
$PROJECTNAME	Inserts the name of the current project on the command line.
$SAVEALL	Saves all changed edit files before starting the program.
$DOCNAME	Inserts the name of the file in the active window on the command line.
$PRIMARY	Inserts the name of the primary file on the command line.

When all boxes are filled, click the *OK* button.

The *Tools Menu Editor* dialog box reappears and the program appears in the *Tools* list. You can then use the *Edit* button to change the set-up for any of the tools, or the *Remove* button to remove a tool. Click the red arrow buttons to move an item up or down in the list. When you are finished, click the *Close* button.

Start a program

To start the program you added, select *Tools* from the menu, and then select the program. Any programs added will be listed at the bottom of the menu.

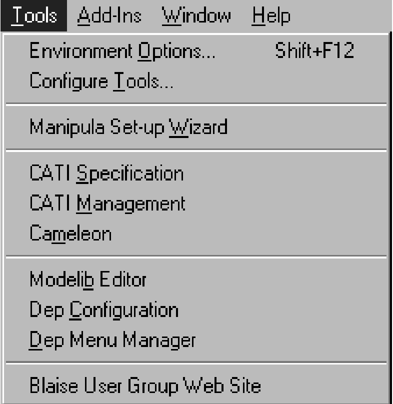
An example: Adding web link

To start a web browser and specify a URL to launch, select *Tools* ➤ *Tools* from the menu and the *Tools Menu Editor* dialog box appears. Complete the box as shown in Figure 2-27.

Figure 2-27: Tools Edit dialog box completed for IBUG web link

Click the *OK* button to return to the *Tools Menu Editor* box and then click the *Close* button. The Blaise User Group Web Site option now appears in the *Tools* menu and can be invoked from the Control Centre, as shown in Figure 2-28.

Figure 2-28: Tools menu with Blaise User Group Web Site link



2.2.10 Manipula Wizard

Manipula is a Blaise tool used to manipulate data and data files. For example, you can create a Manipula file that will transfer data from an input file to an output file.

The Manipula Wizard allows you to create simple Manipula files quickly and easily. Using the Wizard, you can create Manipula files to transfer from:

- Blaise to ASCII
- Blaise to ASCIIRelational
- Blaise to Blaise

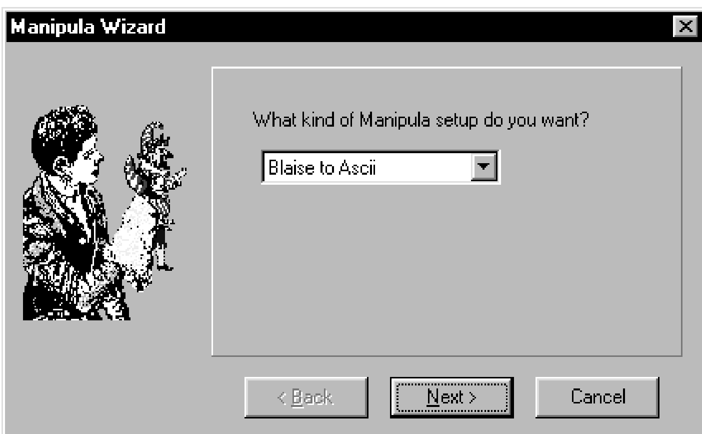
- ASCII to ASCII
- ASCII to Blaise
- ASCIIRelational to Blaise
- OleDb to ASCII (see Section 2.2.14)
- OleDb to Blaise (see Section 2.2.14)

Manipula is discussed in great detail in Chapters 7 and 8. This section describes how to create a simple Manipula setup file using the Manipula Setup Wizard.

Run the Wizard

To run the Manipula Setup Wizard, select *Tools ➢ Manipula Setup Wizard* from the menu. The wizard dialog box appears.

Figure 2-29: Manipula Setup Wizard dialog box



The Wizard displays a series of dialog boxes that prompt you for information. When you complete each box, click the *Next* button to continue. Each step is described below.

Indicate the type of Manipula setup. Click the down arrow to the right of the box to see all the options.

Specify the input data file name and the meta file name.

Specify the type of records you want to include in the output file. *Clean* records are error-free. *Suspect* records have soft errors that are not suppressed. *Dirty*

records have hard errors. *NotChecked* includes records that were not checked for cleanliness. The default is to include all records.

Specify whether you want to copy or move the records from the input data file to the output data file. If you choose to move the records, they will be deleted from the input file. The default is to copy the records.

Specify the name of the output data file and the name of the meta file. If you are transferring to an ASCII file, be sure to indicate a meaningful file extension, such as *.asc*. If you are transferring to a Blaise data file, the file extension will be automatically assigned.

Specify if you want to create a new output data file or append to an existing output data file of the same name. If you already have an output file of the same name and choose to create a new file, the original output file will be deleted. Be sure this is what you want to do.

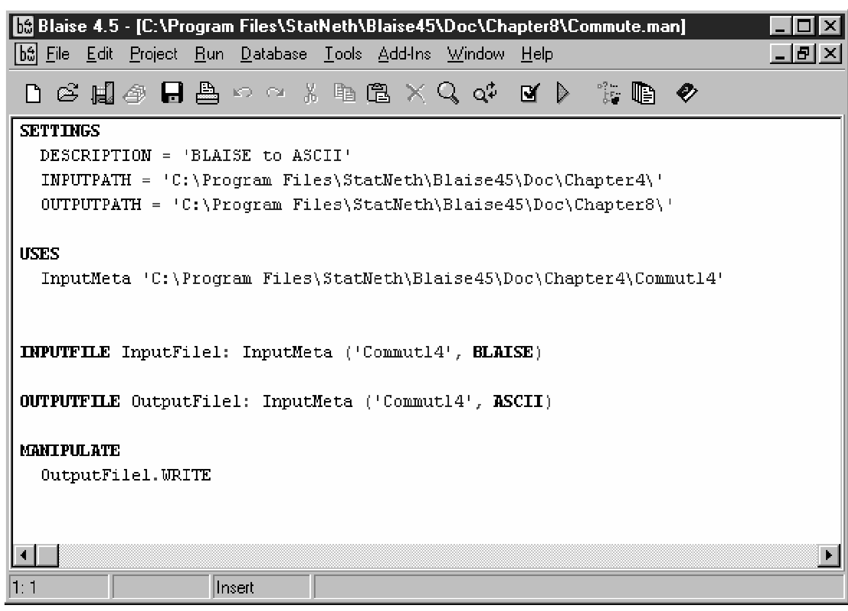
If the output file is to be an ASCII file, indicate if you want a character to separate the fields. This is optional. To specify a character, click the *Field separator specified* box, then choose a character from the list. If you specify a field separator, you also have the option to specify a string field delimiter by clicking the box and selecting a character.

Specify a file name for the Manipula file. The Wizard will generate this file but you must specify a name for it.

Click the *Finish* button.

Blaise creates the Manipula setup file and the file opens in the Control Centre. The following sample is a Manipula file created to convert a Blaise data file to an ASCII data file.

Figure 2-30: Sample Manipula file created with Manipula Wizard



You can use the text editor to modify the Manipula file if necessary.

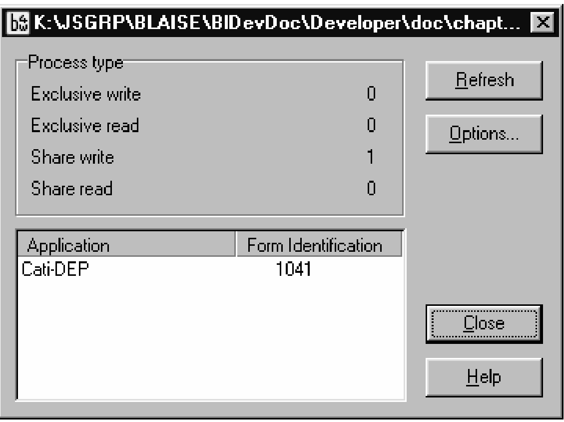
Many Manipula setups are more complicated than this. Nevertheless, using the Manipula Wizard is a good way to start any Manipula setup. For information on creating and running Manipula files and their results, refer to Chapters 7 and 8.

2.2.11 Monitor utility

The Monitor utility can be used to monitor Blaise data files. This is particularly useful in a network environment where multiple users are working on the same data model simultaneously. Monitor provides a real-time summary of who is accessing the data file.

To run Monitor, select *Database* ➤ *Monitor usage*, or run the program `monitor.exe`, which is in the Blaise system folder. Open a Blaise data file and the *Monitor form* appears.

Figure 2-31: Monitor form



This box shows the number of different processes currently active on the data file.

A distinction is made between exclusive write, exclusive read, share write, and share read processes.

- An exclusive write process has exclusive access to the database for both writing and reading. Such a process is active when Manipula is updating a Blaise database in the default access mode.
- An exclusive read process has exclusive access to the database for reading only. Such a process is active when Manipula is processing an input Blaise database in the default access mode or when an external file for the Data Entry Program (DEP) has a read-only flag.
- A share write process has shared access to the database for writing and reading. Such a process is active during a data entry session.
- A share read process has shared access to the database for reading only. Such a process would be active on a database that is used as an external file in the DEP.

Only one application can get exclusive write access, and when this occurs, other applications cannot get access. Multiple applications can get exclusive read access (such as DEP external files that are read-only), but only if no other access rights have already been granted. Exclusive write/read access is fast, because no opening and closing is required for each access.

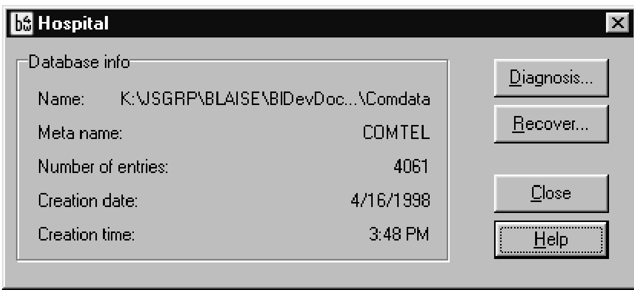
Monitor automatically checks the status and refreshes the window every 10 seconds by default. You can change this by clicking the *Options* button. You can refresh Monitor yourself by clicking the *Refresh* button.

2.2.12 Hospital utility

The Hospital utility checks the integrity of Blaise data files. It can also rebuild corrupted files.

To run Hospital, select *Database* ➢ *Hospital* from the menu, or run the program `hospital.exe`, which is in the Blaise system folder. Open a Blaise data file and the *Hospital* form appears.

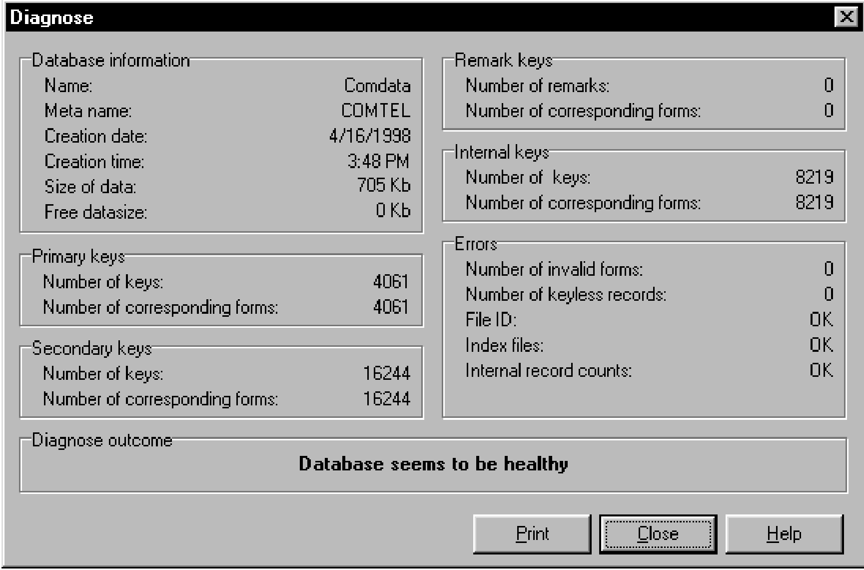
Figure 2-32: Hospital form



This box shows the name of the data file, the metadata file name, the number of forms, and the creation date and time.

To diagnose the file, click the *Diagnosis* button. Hospital checks the file. If the file is healthy, the following diagnosis completion message appears.

Figure 2-33: Diagnose completion message



This box summarises information about the data file. Note the *Diagnose outcome* at the bottom of the box.

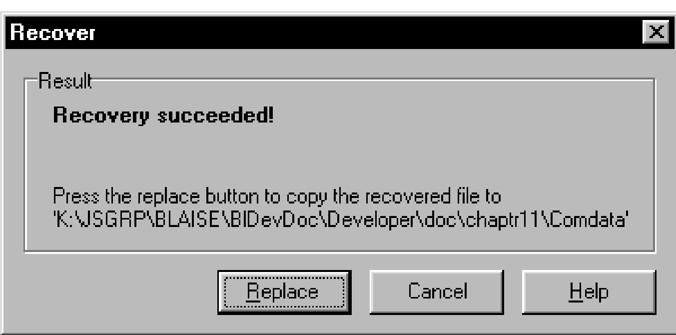
If the data file is not healthy, the system displays a message. Many things could cause a data file to be unhealthy. The sample below shows the message that appears if a primary key file is missing.

Figure 2-34: Diagnose completion message

Click the *OK* button and the *Hospital* dialog box appears.

To rebuild the data file, click the *Recover* button on the *Hospital* dialog box. The system attempts to rebuild the data file. When it has been successfully completed, the following message appears:

Figure 2-35: Successful recovery in Hospital



If you want to replace the old data file with the recovered data file, click the *Replace* button. Hospital creates a new data file with the same name, and a backup of the old data file, with a `.!bd` extension.

2.2.13 Command line prepare utility

Blaise provides a command line prepare utility that can be used for batch prepare of a data model or Manipula/Maniplus setup. This program is a console application, and therefore, it does not show a window on the screen.

Run this utility by invoking `B4CPars.exe`, which can be found in the Blaise system folder. Command line options for this utility are listed in Appendix A.

2.2.14 A remark on OleDB

It is now possible to access relational databases from within Blaise by using the Blaise OleDB interface. This functionality is only available when the Blaise Component Pack has been installed.

To access a relational database from within Blaise, create a Blaise OleDB interface file (a BOI file). You can do this by activating the Blaise OleDB wizard in the Blaise Control Centre. If the Wizard is not present there, it has not been installed on your computer.

An important task of the wizard is to map the fields present in a table or view in your relational database to fields defined in a Blaise data model. The wizard supports two ways to map the fields. The first way is by generating a Blaise data model based on the meta information that can be extracted from the table or view. In this case each field (column) present in the table/view will be mapped to a field in the generated data model. The second way is to provide an already existing

data model and to map the fields (some or all) of the table/view to fields in that data model. The result of the Wizard is always a BOI file that contains the mapping between columns in the table/view and fields in a data model.

2.3 Structure Browser

When you create and prepare data models, it is often useful to review their overall structure and associated information. There are two methods of doing this, each with its own advantages. The first is to use Cameleon, Blaise's metadata utility. It can produce diagrams of the block structure of the data model, as well as other descriptive information, which can be saved to a file and then viewed or printed.

A more dynamic way to review the structure is through the Structure Browser. The Structure Browser provides an overview of the data model. You can see the relationship of fields, blocks, nested blocks, and other Blaise items. You can focus on part of a data model and explore hierarchies, and you can control the type of information displayed. The Structure Browser is an excellent tool for understanding data models, especially ones with which you are unfamiliar.

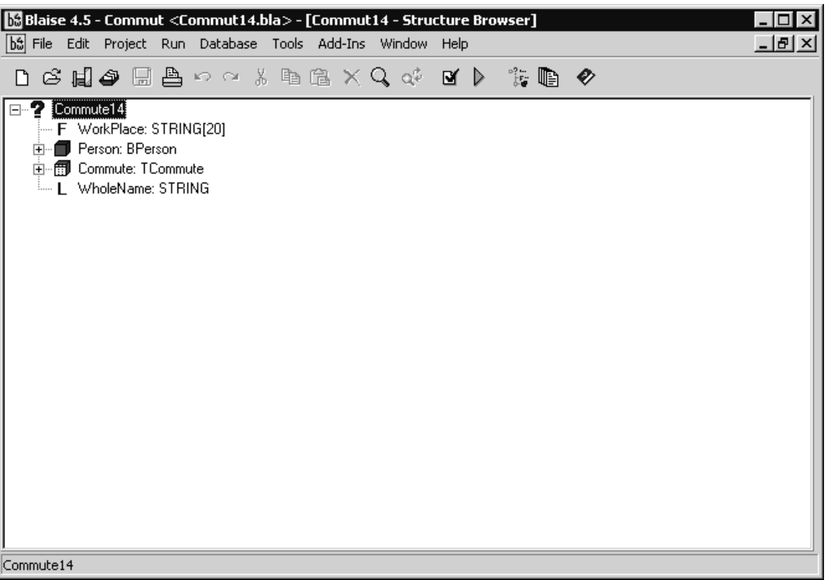
2.3.1 Viewing the structure

The Browser views prepared instrument files, not the Blaise source code file. Therefore, you have to prepare the data model before you can view its structure. It is the preparation that creates the metadata file you will be viewing.

After preparing, select *Database ➤ Browse Structure* from the menu, or click the Browse Structure speed button on the Speedbar. Select a metadata file with a `.bmi` extension.

The Structure Browser opens in the Control Centre. The following sample shows the Structure Browser with the file `commut14.bmi`, which is the prepared file for the `commut14.bla` data model:

Figure 2-36: Structure Browser



The browser shows a hierarchical tree of the data model, with the data model name at the top of the tree. The fields appear in the order in which they are specified in the data model, not necessarily in the order that they appear when the instrument is run.

Blaise displays an icon or a letter next to the items in the tree. Figure 2-37 lists the icons and letters and their descriptions.

Figure 2-37: Icons and letters in the Structure Browser

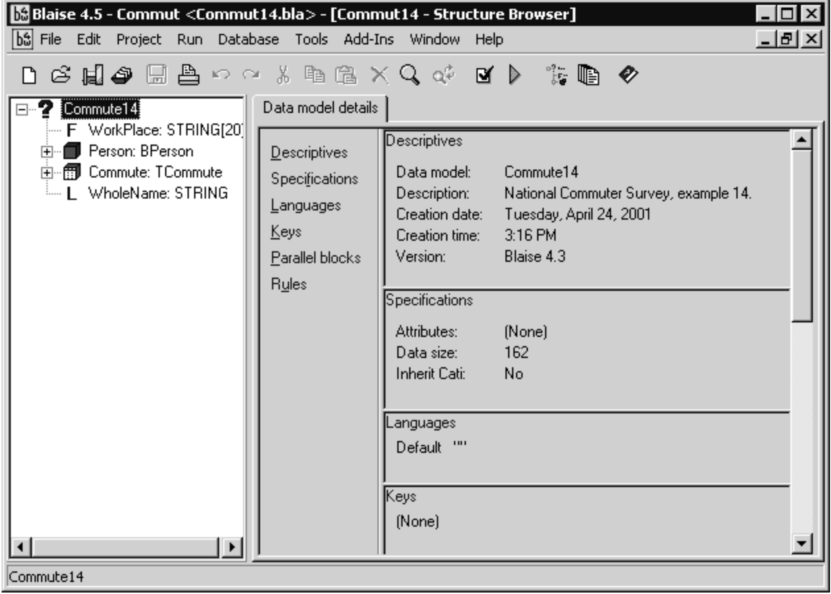
Icon/Letter	Description
?	The top level of the data model.
	Block
	Embedded block
	Array of blocks.
	Array of embedded blocks
	Table
	Embedded table
	Array of tables
	Array of embedded tables
F	Field
A	Auxfield
E	External
L	Local
P	Parameter
ga	Generated auxfield
gp	Generated parameter

Expand and collapse branches of the data model tree by clicking the plus sign or minus sign that is next to an array, block, or the data model name. You can also use the left and right arrow keys.

Detail panel

To see more details of the model, right-click on the Structure Browser window and select the *Details* option from the pop-up menu, or select *Database ➢ Show Details* from the menu. The *Data model details* panel appears on the right side of the window.

Figure 2-38: Structure Browser with detail panel



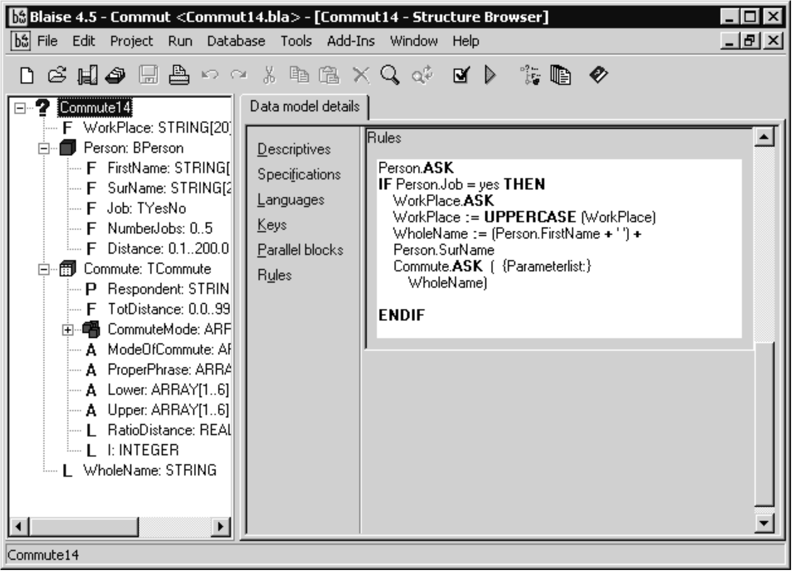
The detail panel shows additional information about the data model. If you click on the uppermost node of the tree on the data model name itself, the detail panel gives you the option to view *Descriptives, Specifications, Languages, Keys, Parallel blocks,* and *Rules* for the model. These are listed on the left part of the detail panel. You can go directly to the section you are interested in by clicking on it. Blaise will move that section of information to the top.

You can resize the panels by placing the mouse pointer on the separator bar that is between the tree and the detail panel and dragging it to another position. You can also use the horizontal and vertical scroll bars to view the tree and the detail panel.

Field, array, and block details

As you click on different parts of the data model tree, the detail panel on the right changes to display specific details about blocks, arrays, or fields. When you click on a field on the tree, the detail panel gives you the option to look at *Descriptives, Specifications, Text,* and *Description text*. If you click on an enumerated field on the tree, Blaise displays the enumeration details on the panel. If you click on an array or block on the tree, you can view the *Rules* for the block or array, as shown in Figure 2-39.

Figure 2-39: Structure Browser with Rules section



You can cut and paste text that appears in the *Rules* and *Text* sections of the detail panel.

Find feature

You can use Blaise's *Find* function to search for specific items in the Structure Browser, on either the tree or the detail panel. Right-click on the area of the window you want to search in, and select the *Find* option from the pop-up menu.

Run from the Structure Browser

You can also run a data model from the Structure Browser. Open the metadata file, and then select *Run* ➢ *Run* from the menu or click on the run icon in the toolbar.

2.3.2 Structure Browser options

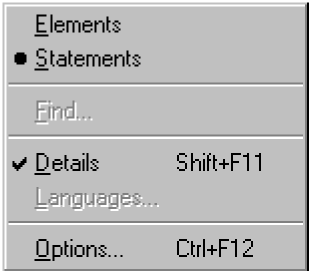
You can set display options for the Structure Browser. Right-click on the Browser window and select *Options* from the pop-up menu. The Structure Viewer options appear in the *Environment Options* dialog box. These can also be set by selecting *Tools* ➢ *Options* from the menu.

Click the appropriate boxes to display the type of fields listed in the *Fields to show* section. To display internal parameters, click the appropriate boxes under the *Internals to show*. Internal parameters are discussed in the section on data model performance in Chapter 5.

2.3.3 Viewing data model statements

In addition to displaying the various components of the data model, the structure browser also can show in a tree display the statements that control the routing of the data model. To see this display, right-click in the left panel and from the pop-up menu select *Statements,* or from the Control Centre menu select *Database ➤ Statements View*.

Figure 2-40: Pop-up menu to display statements



Now the left-side panel is transformed into a display of the data model statements, showing information for each block, question, computation, conditional, edit check, and other objects used in the Rules.

Blaise displays an icon next to the items in the tree. The following table lists the icons and their descriptions.

Figure 2-41: Blaise icons with descriptions

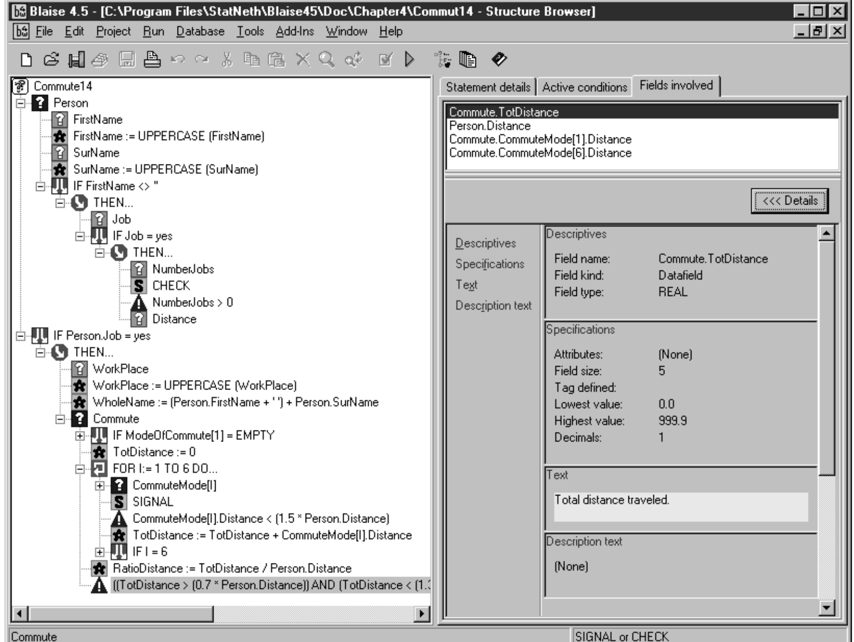
Icon/Letter	Description
	The top level of rules.
	Field Ask
	Field Show
	Field Keep
	Block Ask
	Block Show
	Block Keep
	For-do structure
S	Statement (for instance SIGNAL, CHECK)
	Assignment / Procedure
	Edit
	If condition
	Then part
	Else/Else-if part

The statement display provides a flexible, view of precisely what is happening in the data model. In developing or analysing a complex Blaise data model with many if-then conditions and routes, iterative loops, edit checks, external files and other elements, the statements view is a powerful tool. Figure 2-42 provides a sample statement display.

Figure 2-42: Structure browser with statement display

The right-side panel has detailed information about the selected statement or object. For example, in the following figure, a signal-type edit check is selected. Along with the conditional expression shown on the left, the right, or details, panel provides in three tabs full information on the statement details, the active conditions and the fields involved.

Figure 2-43: Browse statement Fields Involved



2.4 Database Browser

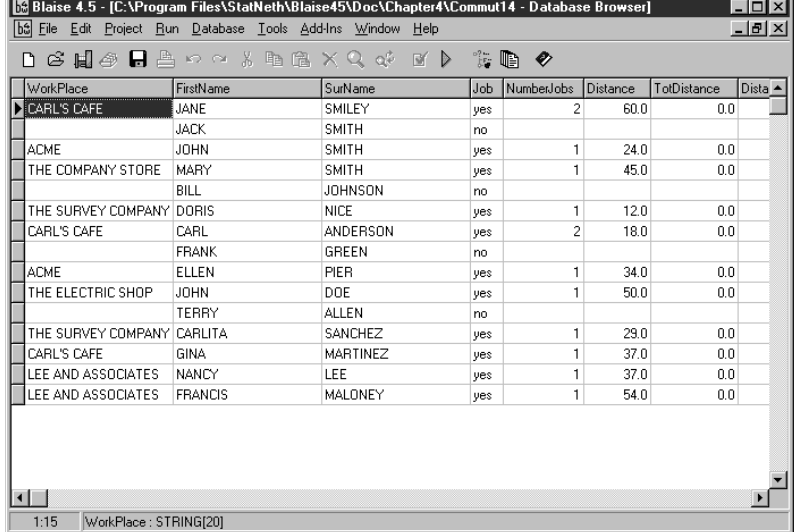
A Blaise data file is not a text file but a binary file. You cannot view it in a different file browser or in a text editor, and you might even corrupt the file if you attempt to do so. Blaise therefore provides a Database Browser that allows you to look at the contents of a Blaise data file. You can browse the data and view details of it, search on key fields, and select and save a specific view of the data.

2.4.1 Viewing the data

To view the data, select *View* ➤ *Browse Contents* from the menu, or click the *View* speed button on the Speedbar. The *Open* dialog box appears. Select a Blaise data file with a `.bdb` extension and the Database Browser window appears.

The sample below shows the data file `commut14.bbd` that was created by running the data model `commut14.bla`, and then entering sample data.

Figure 2-44: Database browser

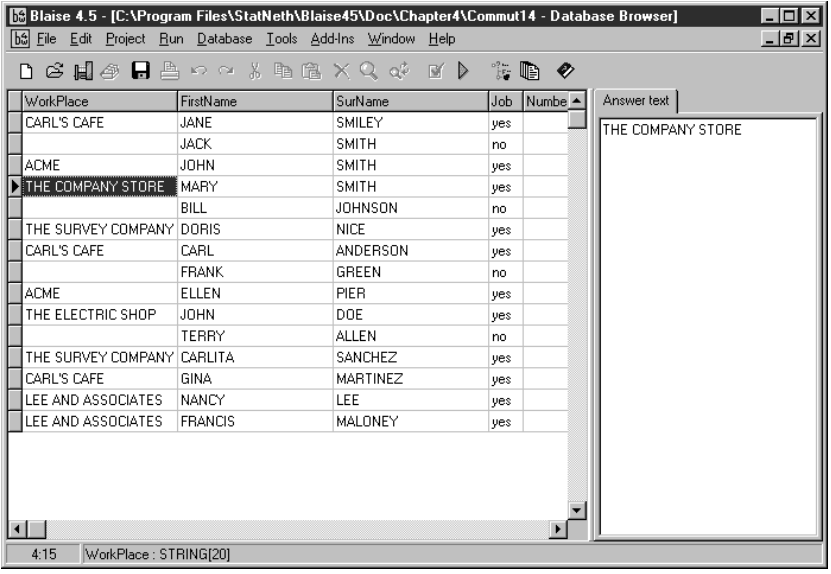


WorkPlace	FirstName	SurName	Job	NumberJobs	Distance	TotDistance	Dista
CARL'S CAFE	JANE	SMILEY	yes	2	60.0	0.0	
	JACK	SMITH	no				
ACME	JOHN	SMITH	yes	1	24.0	0.0	
THE COMPANY STORE	MARY	SMITH	yes	1	45.0	0.0	
	BILL	JOHNSON	no				
THE SURVEY COMPANY	DORIS	NICE	yes	1	12.0	0.0	
CARL'S CAFE	CARL	ANDERSON	yes	2	18.0	0.0	
	FRANK	GREEN	no				
ACME	ELLEN	PIER	yes	1	34.0	0.0	
THE ELECTRIC SHOP	JOHN	DOE	yes	1	50.0	0.0	
	TERRY	ALLEN	no				
THE SURVEY COMPANY	CARLITA	SANCHEZ	yes	1	29.0	0.0	
CARL'S CAFE	GINA	MARTINEZ	yes	1	37.0	0.0	
LEE AND ASSOCIATES	NANCY	LEE	yes	1	37.0	0.0	
LEE AND ASSOCIATES	FRANCIS	MALONEY	yes	1	54.0	0.0	

Detail panel

To see more details, right-click on the window and select the *Details* option from the pop-up menu, or select *Database* ➤ *Show Details* from the menu. A detail panel appears on the right, as shown in Figure 2-45.

Figure 2-45: Database browser with detail panel



The detail panel displays the answer text for the fields in your data file. If a field contains a remark, that is also displayed on the detail panel with a separate *Remarks* tab. You can also cut and paste text from the detail panel.

Resize and reposition columns

As with the Structure Browser, you can resize the panels by placing the mouse pointer on the separator bar and dragging it to another position.

Resize a column by placing the mouse pointer on the separator between columns and dragging it. You can set specific column widths by setting Database Browser options; these are discussed in the following section. Reposition a column by clicking on it and dragging it to a new location.

Edit column titles

You can change the column titles by right-clicking on the column title to be changed. The *Change Header* dialog box appears. Enter a new title and click the *OK* button, and the new title appears for that column.

Search on key fields

If you have identified key fields in the data model, a *Search* box and a *Key type* box appear at the bottom of the Database Browser window.

Figure 2-46: Key type box

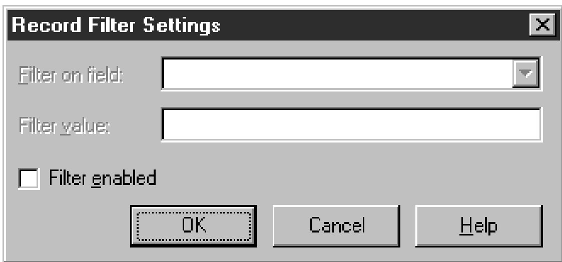
- To search for specific responses using the keys, click the arrow in the *Key type* box to select a key type, then type the text to search in the *Search* box.
- You can search for the values in the format in which they are stored in the Blaise database or in the format in which the fields are defined. For example, for enumerated or set fields, you can search for the database value (such as *1, 2,* and so on) or for the value label of the response (such as *yes* or *no*).
- To search using the formats of the field definitions, you must check the Smarter search option in the Environment Options dialog box, Database Browser tab. (See the following section, 2.4.2 Database Browser options.)
- For keys that contain an enumerated field or a set field, if *Smarter search* is not checked, enter the database value of the field (such as 1, 2). If *Smarter search* is checked, enter the label of the response, such as Yes or No.
- For keys that contain a numeric field, if *Smarter search* is not checked, enter the value followed by a semicolon (;). If *Smarter search* is checked, just enter the actual value.
- For keys that contain a string field, search for the actual value of the field.
- For keys made of multiple fields, separate the values to search on with a semicolon. This applies whether or not *Smarter search* is checked.
- For example, suppose you have a secondary key that is made of the fields *Job* and *Sports,* and both fields are enumerated with *Yes* and *No* responses. To search for the database value of a form that has *Job = Yes* and *Sports = No*, you would enter *1;2* in the *Search* box. To search for the value label, you would first make sure the *Smarter search* option is checked, and then you would enter *Yes,No* in the *Search* box.
- In another example, suppose you have a date field, and your Windows® date settings are in the format *mmddyyyy.* Date fields are stored in the Blaise database in the format *yyyymmdd.* If you want to search for the date *12011998* and *Smarter search* is checked, and you would enter *12011998* in the *Search* box. Otherwise, you would enter *19981201*.

Keys are discussed in more detail in Chapters 3 and 5.

Record filter

If secondary keys have been identified in your data model, you can filter the data based on the secondary keys. Select *Database* ➢ *Record Filter* from the menu, or right-click on the window and select the *Record Filter* option. The *Record Filter Settings* dialog box appears.

Figure 2-47: Record Filter Settings dialog box



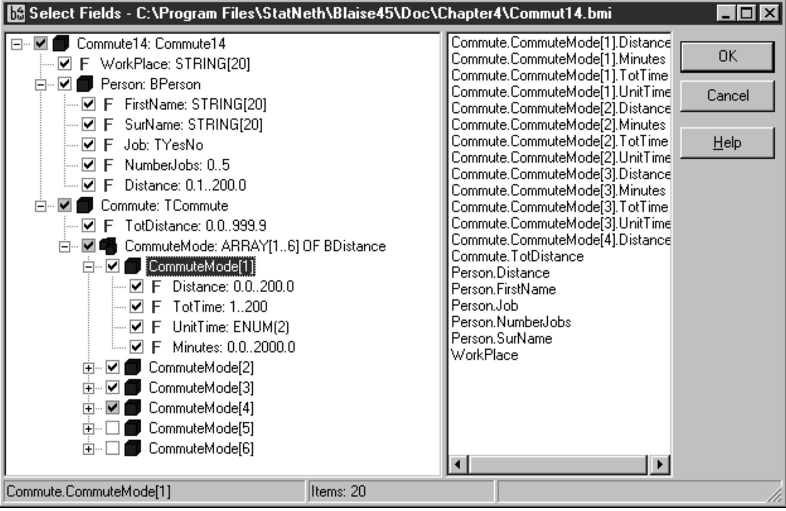
To enable the filter, click the *Filter enabled* box. Select a field from the *Filter on field* box and type a value for the field in the *Filter value* box. Click the *OK* button.

The browser displays only those records that match the value input for that key.

Select fields to view

You can select specific fields to view in the Database Browser. Select *Database* ➢ *Select Fields* from the menu, or right-click on the Database Browser window to display a pop-up menu. The *Select Fields* dialog box appears.

Figure 2-48: Select fields dialog box



The structure of the data model is on the left and the selected fields are on the right. Expand and collapse the tree by clicking the plus signs or minus signs or using the left and right arrow keys. Double-click in the box next to each block, array, or field that you want displayed, or use the space bar to toggle the check on and off.

For blocks, the box next to the block field can appear three different ways. If the box is empty, no fields within the block are selected. If the box is grey with a check mark, some fields in the block are selected. If the box is white with a check mark, all fields in the block are selected.

When you are finished, click the *OK* button.

The Database Browser window reappears and shows only those fields that you selected.

Save a view

You can save a specific view of your data file. This is useful if you want to show only specific parts of a data file to others, or if you are debugging an instrument and want to repeatedly view only parts of the data file.

First, select the fields you want in the view by clicking on them. Then select *File ➤ Save As* from the menu. A *Save Field Selection As* dialog box appears. Save the view settings in a file; the default file extension is `.bdv`. You can then open this file in the Database Browser.

Running the DEP from the database browser

You can also run a data model from the Database browser. Highlight in the database browser the record you wish to open and click on the run icon in the toolbar or select Run ➤ Run from the menu. The DEP will be run on this record.

2.4.2 Database Browser options

You can set display options for the Database Browser. Right-click on the window and select *Options* from the pop-up menu. The Data Viewer options appear in the *Environment Options* dialog box. These can also be set by selecting *Tools ➤ Options* from the menu.

- *Code names.* Select to display the name instead of the code for enumerated fields.
- *Field type.* Select to display the field type, such as string or enumerated, on the status bar.
- *Key fields.* Select to display the key fields in the first columns of the Browser. By default this option is checked. Key fields will only be displayed when this option is selected.
- *Smarter search.* Select this option to take into account the field definitions of the key when searching. If this option is not checked and you search on key fields in the Database Browser, you have to search on the database value of the field in the format that it is stored in the database. For example, date fields are stored in the Blaise database in the format *19981201*. If you select this option, you can search for the date in the format that corresponds to your Windows® date settings (for example, as *12011998*). This option applies to all field types. (See section *2.4.1 Viewing the data*, *Search on keys* for more examples.)
- *Internal record number.* Select to display a sequential record number.
- *Correctness status.* Select to display whether the records are clean, dirty, suspect, or not checked.
- *Number of errors.* Select to display the number of errors in each record.

- In the *Column width* section, click *Default* to set the column width equal to the field name size. Click *Data width* to set the column width equal to the length of the data only. To set a specific column width, type a number in the *Maximum column width* box. This is most useful for string, memo, and classification fields. To set a specific number of fields to display, type a number in the *Maximum fields to display* box.
- The *Font* and *Size* boxes change the display font used.

2.5 Help

Blaise provides extensive on-line help. When you select *Help* from the menu, you have two options. *Help topics* contains on-line help on the Blaise system. *Reference manual* contains specific Blaise language information. You can also access context-sensitive help by pressing F1.

2.5.1 What's New

In this help topic all information present in the readme files of all releases since the first Blaise version 4.0 release can be accessed. This provides easy access to the many features and capabilities of Blaise. In addition the What's new topic discusses other important topics, including:

- Information for Blaise III users
- Environment settings
- Manipula Configuration File
- Translating Blaise to a different language
- Blaise Help Files
- Oem to Ansi (upgrading from version 4.2 or earlier to version 4.3 or higher)
- Blaise Command Line Option File

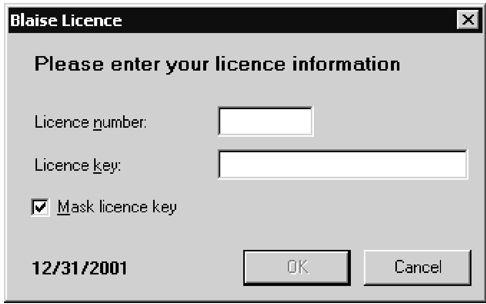
2.5.2 Enter registration

Help|Enter registration opens a dialog box into which you can enter the license information for your Blaise development system. Typically this is only used when a licensed is renewed or updated to enable an extra capability.

Enter the license number and license key provided. The *Mask license key* check box, when enabled, displays '*' characters as you type in the license key value.

The current expiration date of the license is shown in the lower left corner.

Figure 2-49: Enter registration dialog box



3 Data Model Basics

A data model in the Blaise® language states the definition and structure of the survey data and their interrelationships. In order to set up a system for collecting the proper data, you must specify the data model in explicit and unambiguous terms. The data model can hold all information that is needed throughout the complete survey process, including routing and consistency relationships between variables. The data model also serves as a knowledge base for all other parts of the system.

The data model specification in the Blaise® language is checked and prepared for production use in the Control Centre. To enter data based on your data model, there is a Data Entry Program, or DEP, that can be used for interviewing, data editing, and data entry. The program is also referred to as an instrument.

This chapter gives an overview of the Blaise language and covers basic Blaise language elements and concepts. With these you can build many useful data models. Blocks and tables, which you can use to build very large and complex data models, are covered thoroughly in Chapter 4. Advanced and specialised topics are covered in Chapter 5.

3.1 Blaise Language Overview

You specify a Blaise data model through the Blaise language. This section provides an overview and a few data model samples. The three data models in this section only give a taste of the language. Much larger and more complex data models are possible. You can prepare the code from these three data models with the files `commute1.bla`, `commute2.bla`, and `commute3.bla`. They are found in `\Doc\Chapter3` in the Blaise system folder.

Field definition

A *field definition* includes a field name and a field type that defines valid values. Usually it will have question text. It can have a description to document the field, a field tag, and special attributes.

Rules

There are four types of rules: routing instructions, edit checks (hard and soft), computations, and layout instructions.

- *Routing instructions* describe the data entry order of the fields and the conditions under which they will be eligible. For computer-assisted interviewing, the route specifies the order and conditions in which the fields are asked.
- *Edit checks* determine whether a specified statement is true for the values of the fields involved. If it is false, the instruction will generate an error. What will be done with the error depends on the application at hand. Two kinds of edits are supported. The CHECK instruction defines a hard error, something that must be fixed before the form can be considered clean. The SIGNAL instruction defines a soft error, which is a possible problem. It can be suppressed or the values of the involved fields may be changed. A CHECK is the default.
- *Computations* determine proper routes to process fields, carry out complex checks, or derive values.
- *Layout instructions* determine the placement of data entry fields displayed in the Data Entry Program.

We will illustrate various aspects of the Blaise language by using a simple example: a survey that investigates the behaviour of commuters. We start with a simple data model. The program code for this data model can be found in `commute1.bla` in `\Doc\Chapter3` of the Blaise system folder.

The population consists of people, and for each person we want to have values for six fields: their name, the town where they live, gender, marital status, number of children (for women only), and age. The following example contains a possible specification of this data model:

```
DATAMODEL Commute1 "The National Commuter Survey, first example."

  FIELDS
    Name     "What is your name?" : STRING[20]
    Town     "In which town do you live?" : STRING[20]
    Gender   "Are you male or female?" : (Male, Female)
    MarStat  "What is your marital status?" :
             (NevMarr  "Never married",
              Married  "Married",
              Divorced "Divorced",
              Widowed  "Widowed")
    Children "How many children have you given birth to?" : 0..10
    Age      "What is your age?" : 0..120

  RULES
    Name
    Town
    Gender
    MarStat
    Age
    CHECK
    IF (Age < 15) "If age is less than 15" THEN
      MarStat = NevMarr
      "then he/she is too young to be married !"
    ENDIF
    IF (Gender = Female) AND (Age >= 12) THEN
      Children
    ENDIF

ENDMODEL
```

Reserved words or key words

Certain words such as DATAMODEL, FIELDS, and ENDIF have special meaning in the Blaise language and are called *reserved words* or *key words.* These words are printed in uppercase in this guide. Their use is reserved for special situations. To emphasise this special meaning in the code examples, they are printed in capitals. In your own programs, reserved words may be typed in lowercase, or in a mixture of lowercase and uppercase letters. It is a good programming practice to type reserved words in a recognisable way. Most key words are highlighted in the Control Centre.

The first line of the specification contains the reserved word DATAMODEL followed by a name and, optionally, a longer explanatory text between double quotes. The end of the specification is indicated by the reserved word ENDMODEL.

Fields

A field is the basic element of the data set in Blaise. Fields can have specific types of definitions. Examples of definition types include strings, numbers, or dates. You can create your own user-defined types of fields.

You specify fields in the FIELDS *section*. In its most simple form, each field definition consists of an identifying name and a specification of the valid values. A longer text between double quotes will usually be inserted between the name and the value definition. This text may serve to state a question, as description, or to document the field.

Identifier

The above sample data model contains six fields. The first two fields are *Name* and *Town*:

```
Name "What is your name?"         : STRING[20]
Town "In which town do you live?" : STRING[20]
```

These are string fields. They can hold any text up to 20 characters. The fields *Children* and *Age* are numeric fields:

```
Children "How many children have you had?" : 0..10
Age      "What is your age?"               : 0..120
```

The values for *Children* have to be within the range from 0 to 10, and the range for *Age* is 0 to 120. The fields *Gender* and *MarStat* are enumerated (also called precoded) fields:

```
Gender  "Are you male or female?" : (Male, Female)
MarStat "What is your marital status?" :
         (NevMarr  "Never married",
          Married  "Married",
          Divorced "Divorced",
          Widowed  "Widowed")
```

There is a list of possible values assigned to each of these two fields. One value has to be picked from each list. The list for *Gender* contains the two items *Male* and *Female*. The list for *MarStat* consists of four items. Each item has an identifying name and an explanatory text.

The second part of the data model specifies the rules that have to be obeyed in processing the data. This section starts with the reserved word RULES. Rules come in three forms: routing instructions, edits, and computations.

Routing

Let us start with the routing instructions. Writing down the name of a field in the RULES section means obtaining a value. For an interviewing program this implies

asking the question. The RULES section of the example starts with the five field names *Name*, *Town*, *Gender*, *MarStat*, and *Age*.

```
RULES
  Name
  Town
  Gender
  MarStat
  Age
```

These five fields will be processed in this order. Field names can also be asked, subject to a condition. For example:

```
IF (Gender = Female) AND (Age > 12) THEN
   Children
ENDIF
```

This means that the field *Children* will only be processed if the field *Gender* has the value *Female* and *Age* is greater than 12.

Checking

Checks are conditions that have to be satisfied. You can state the check in terms of what the correct relationship between fields should be.

```
MarStat = NevMarr "he/she is too young to be married!"
```

The specification instructs the system to check whether the field *MarStat* has the value *NevMarr*. If not, then the edit is invoked. You can attach text between double quotes to a condition. Such a text will be used as an error message if the condition is not satisfied.

Checks can be subject to conditions:

```
IF (Age < 15)
   "If age of respondent is less than 15" THEN
    MarStat = NevMarr
   "then he/she is too young to be married!"
ENDIF
```

The check *MarStat = NevMarr* will only be carried out if the field *Age* has a value less than 15. The application will reject entries in which people younger than 15 years are married.

As an alternative to *Checks,* the *ERROR* function can be used in cases where the result of the condition is always false. *ERROR* allows you to generate an error after a complex IF...THEN structure of checks, in a branch which should logically not be reached. In other cases, its use is not advised.

An error check is not directly related to fields. It is best to use the *INVOLVING* instruction to activate edit fields to avoid situations in which you cannot repair the error.

```
IF Age<15 INVOLVING (BDate) THEN
  MarStat = NevMarr
ELSEIF MarStat = Married THEN ...
ELSEIF Children = 0 THEN
  ...
ELSE
  ERROR "One or more children expected"
ENDIF
```

The field, *Children,* is the only field involved in the error check.

```
DATAMODEL Commute1 "The National Commuter Survey, first example."

  FIELDS
    Name     "What is your name?" : STRING[20]
    Town     "In which town do you live?" : STRING[20]
    Gender   "Are you male or female?" : (Male, Female)
    MarStat  "What is your marital status?" :
             (NevMarr  "Never married",
              Married  "Married",
              Divorced "Divorced",
              Widowed  "Widowed")
    Children "How many children have you given birth to?" : 0..10
    Age      "What is your age?" : 0..120

  RULES
    Name
    Town
    Gender
    MarStat
    Age
    CHECK
    IF (Age < 15) "If age is less than 15" THEN
      MarStat = NevMarr
      "then he/she is too young to be married !"
    ENDIF
    IF (Gender = Female) AND (Age >= 12) THEN
      Children
    ENDIF

ENDMODEL
```

The complete `commute1.bla` data model, shown above, has six fields that deal with personal characteristics and not with commuter behaviour. We will now extend the data model with five fields that contain information about commuting. To keep the fields in the model well organised, we will distribute the fields over two blocks: one with personal characteristics and one with information about work and commuting.

The program code for this data model can be found in the file `commute2.bla` in `\Doc\Chapter3` under the Blaise system folder.

```
DATAMODEL Commute2 "The National Commuter Survey, example 2."

BLOCK BPerson "Demographic data of respondent"

  FIELDS
    Name     "What is your name?" : STRING[20]
    Town     "In which town do you live?" : STRING[20]
    Gender   "Are you male or female?" : (Male, Female)
    MarStat  "What is your marital status?" :
                (NevMarr  "Never married",
                 Married  "Married",
                 Divorced "Divorced",
                 Widowed  "Widowed")
    Children "How many children have you given birth to?" : 0..10
    Age      "What is your age?" : 0..120

  RULES
    Name Town Gender MarStat
    Age
    IF (Age < 15)
          "If age of respondent is less than 15" THEN
      MarStat = NevMarr
         "then he/she is too young to be married !"
    ENDIF
    IF (Gender = Female) AND (Age > 12) THEN
      Children
    ENDIF
ENDBLOCK


BLOCK BWork "Data about work and commuting"

  FIELDS
    Working  "Do you have a paid job?" : (Yes, No)
    Descrip  "Short description of your job" : STRING[40]
    Distance "What is the distance to your work (in km)?" :  0..300
    Travel "How do you travel to your work?" : SET [3] OF
       (NoTravel "Do not travel, work at home",
        PubTrans "Public bus, tram or metro",
        Train    "Train",
        Car      "Car or motor cycle",
        Bicycle  "Bicycle",
        Walk     "Walk",
        Other    "Other means of transportation")
    Commuter "Are you a commuter?" : (Yes, No)

  RULES
    Working
    IF Working = Yes THEN
      Descrip Distance Travel
    ENDIF
    IF (Working = Yes) and (Distance > 10) THEN
      Commuter := Yes
    ELSE
      Commuter := No
    ENDIF
ENDBLOCK

FIELDS
  Person : BPerson
  Work : BWork;

RULES
  Person
  Work

ENDMODEL
```

The specification contains two block definitions: the block *BPerson* and the block *BWork*. A block is like a sub-data model with fields and rules. A block is a special kind of field type. You define a field with the block name as field type.

```
FIELDS
  Person: BPerson
```

This is a field of the type *Bperson*, which is a block of fields and rules. Writing down the name of such a field in a RULES section means processing all fields and rules inside the block. The reason that a block is renamed in a FIELDS section as shown above will become clear in Chapter 4. You will find it is a powerful way to repeat blocks of code with just a few words.

The preceding example also contains a compute instruction. Take a look at the field *Commuter*. It is defined in the FIELDS section of the block *BWork*.

The field name appears in the following part of the RULES section:

```
IF (Working = Yes) and (Distance > 10) THEN
    Commuter := Yes
ELSE
    Commuter := No
ENDIF
```

The field *Commuter* gets the value *Yes* if a person's activity is *Working* and the distance to work is more than 10 km. In all other cases, the field *Commuter* will get the value *No*. Note that the interview program based on this model will never ask the question *Commuter*. Instead its answer will always be computed. In fact, it is not even displayed on the screen.

The following example contains a small hierarchical data model. The data model has two levels. The highest level is the household with only three fields: the address of the household (street and town) and the size of the household. In each household there are persons. Each person has a set of demographic fields and a set of fields relating to work and commuting. The following program code can be found in `commute3.bla` in `\Doc\Chapter3` in the Blaise system folder:

```
DATAMODEL Commute3 "The National Commuter Survey, example 3."

TYPE
  TYesNo = (Yes, No)
BLOCK BPerson "Demographic data of respondent"
  FIELDS
    Name     "What is your name?" : STRING[20]
    Gender   "Are you male or female?" : (Male, Female)
    Age      "What is your age?" : 0..120
    MarStat  "Are you married?" : TYesNo;
  RULES
    Name Gender Age
    IF Age >= 12 THEN
      MarStat
    ENDIF
    IF (Age < 15)
       "If age of respondent is less than 15"
        THEN MarStat = No
       "then he/she is too young to be married !"
    ENDIF
ENDBLOCK
BLOCK BWork "Data about work"
  FIELDS
    Working  "Do you have a paid job?" : TYesNo
    Descrip
      "Give a short description of your job" : STRING[40]
    Distance
      "What is the distance (in km) from home to work?" : 0..300
    Travel   "How do you travel to your work?" :SET[3] OF
     (NoTravel "Do not travel, work at home",
      PubTrans "Public bus, tram or metro",
      Train    "Train",
      Car      "Car or motor cycle",
      Bicycle  "Bicycle",
      Walk     "Walk",
      Other    "Other means of transportation")
    Commuter "Are you a commuter?" : TYesNo
  RULES
    Working
    IF Working = Yes THEN
       Descrip Distance Travel
    ENDIF
    IF (Working = Yes) and (Distance > 10) THEN
       Commuter := Yes
    ELSE
       Commuter := No
    ENDIF
ENDBLOCK
LOCALS
  I: INTEGER
FIELDS
  Street "Address of the household, @/Street and number?" : STRING[20]
  Town   "Address of the household, @/Town?" : STRING[20]
  HHSize "Number of persons in the household?" : 1..10
  Person: ARRAY[1..10] OF BPerson
  Work:   ARRAY[1..10] of BWork;
RULES
  Street Town HHSize
  FOR I := 1 TO HHSize DO
    Person[I]
    Work[I]
  ENDDO
ENDMODEL
```

The example shows a number of new elements you can use in Blaise. It has a section for declaring local variables. This section starts with the reserved word LOCALS. You use locals in computations as intermediate variables, or as an index for running through an array of fields. Local variables are of a temporary nature. Their values are not stored.

There is a section that starts with the reserved word TYPE. You can define your own field types in this section. This is particularly useful if you have several fields to which you want to assign the same set of valid values. The hierarchical data model above contains a definition of the field type *TYesNo* that is used in a field definition for the fields *Married*, *Working,* and *Commuter*.

There are two block definitions in the model: *BPerson* and *BWork*. The FIELDS section at the highest level (the household level) contains the following line:

```
Person: ARRAY[1..10] OF BPerson
```

This statement defines a series of 10 fields *Person[1], Person[2], ..., Person[10]* of the block type *BPerson*. All 10 fields represent a set of fields: the fields in the block *BPerson* (*Name*, *Gender*, *MarStat,* and *Age*). Likewise, there is a series of block fields *Work[1]* to *Work[10]*.

The data model shown above is a simple example of a hierarchical data model. There is a series of fields at the household level, and there are fields at the person level. The fields at the person level can be repeated at most 10 times, once for each member in the household.

Programmer's comments

Some organisations consider it a good programming practice to make frequent comments in the Blaise code so that you or your successor can easily see what logic you are implementing. This is done with the braces '{' and '}'. There is no practical limit on the length of comments. You can nest comments. During preparation, if Blaise sees a left brace '{', then everything after that brace will be considered a comment until it sees a right brace '}'.

3.2 Fields

Fields represent the variables to be measured in the survey. The definition should be such that the fields can hold all correct values of the variables that may be encountered in practice. The definition should be strict enough to enable the

system to detect incorrect values, but not values that are merely unlikely. You check unlikely values with a signal, or soft edit, as described in the text that follows.

Fields section

You define fields in a special section of the data model. This section must start with the reserved word FIELDS. All field definitions follow the same scheme. In its simplest form, the field definition is:

```
FIELDS
  Fieldname: FieldType
```

Field names

The field name is specified first. You can use characters from different alphabets, like the 'å' and the 'ö.' Field names can be up to 255 characters, but we suggest you avoid using very long names. Some examples of proper field names are:

```
Head_of_the_household

Year_1

_12_Months

ålder

ÃÄÄ

Année_de_naissance
```

The field name is used in several ways:

- It is used in the RULES section to refer to the field in routing instructions, in IF conditions, computations, and edit checks.
- It is may be used to identify the question in the FormPane on the screen.
- It may be used to list fields to jump to for the interviewer or editor when an edit is invoked.
- It may be used in Cameleon set-ups or the Blaise API to provide documentation or to create data set-up programs for packages like SAS and SPSS.
- It is used in Manipula set-ups.

Note that the field description can be used in the form pane. The use of the field name, with respect to the field description is discussed in the text to follow.

The choice of a good field name is extremely important, especially for the interviewer. Because field names will be displayed to the interviewer if she encounters an edit (meaning that something in the field was input inappropriately), the field name needs to be clear and sensible. For example, the field name *FirstName* is much clearer than *Name1* or *Q1*.

Field types

The second element of the field definition is the field type. The field type tells the system which values to accept. Blaise has several pre-defined field types. They are discussed further in the following sections. A few examples follow:

```
Name : STRING[30]

Age : 0..120
```

Field text

You can include text between double quotes for questions, instructions, or other text displays:

```
FieldName "Text" : FieldType
```

This text is used by the interviewer to ask questions. The maximum length of the text is 32,767 characters. An example:

```
Name "What is your name?" : STRING[30]

Age "How old are you?" : 0..120
```

The metadata program Cameleon can use the text as a field description when it creates set-ups for other packages like SPSS, Oracle, and SAS. It can also use the field description, which is described next.

Field description

The survey process will often consist of several activities. Therefore, you may need different texts for different purposes: a question text for collecting the data, an explanatory text for tabulating data, and variable labels for analysing the data with a statistical package. For this purpose you can attach two texts to a field. The

second text is called a description. The two texts are separated by a slash (/). The field definition scheme can now be extended to the following form:

```
FieldName "Text" / "Description" : FieldType
```

The FieldDescription may be displayed in the Blaise page as an alternative to the *FieldName*. It can also be displayed in the *Edit Jump dialog*. See section 3.6.4. FieldDescription can be used as the interviewer identifier and additionally, as a label in a downstream system for all spoken languages.

When the interviewer switches languages, many things on the screen, including field texts, switch to the active language as well. If there is an edit, then the fields are identified by the readable FieldDescription in the appropriate spoken language. For example:

```
Name "What is your name?" / "Respondent name" : STRING[30]

Age "How old are you?" / "Respondent age" : 0..120
```

Languages

Blaise supports the use of different languages. This is particularly important if you want to interview people speaking different languages. During interviewing, the interviewer can change to a different language with a function key. This requires you to specify the question texts in all languages you might intend to use. The format for a field definition for two languages is as follows:

```
FieldName "Text1" "Text2" /
          "Description1" "Description2" : FieldType
```

You specify the question texts (*Text1* and *Text2*) in different languages, and specify the descriptions after the slash in the same order. Here is a concrete example:

```
Name "What is your name?" "Wat is uw naam?" /
     "Name of respondent" "Naam van de respondent" : STRING[30]
```

Note that you have to specify the languages of your multilingual data model in the SETTINGS section of the data model. See *Section 3.7.2 Languages* for a description of spoken and non-spoken languages.

Field tags

Questions on paper questionnaires are often identified using numbers. We do not advocate this principle. We think that working with and talking about questionnaires is much easier if you give names to questions. It will be much clearer if you talk about the questions *Age* and *Work* instead of talking about questions *Q001* and *Q234*. It also makes maintenance much easier, for instance, if new questions are inserted between numbers *Q001* and *Q002*. Still, when entering data from paper forms, data entry operators like to have an easy way to associate the fields on the form with the fields on the screen. Blaise offers field tags for this purpose.

A field tag can be displayed on the screen and you can jump to fields with a specific field tag. By default, the search for the tag will be conducted from the start of the form, though you can specify the search to start from the current point. The tag search is case-insensitive. A field tag is specified between parentheses after the field name and before the first text:

```
FieldName (FieldTag) "Text" / "Description" : FieldType
```

A field tag may consist of letters and digits, although usually only digits will be used (to represent question numbers). The following example illustrates the use of field tags:

```
Working (101) "Do you have a paid job?" : (Yes, No)

Descrip (102) "What is your job description" : STRING[40]
```

It is possible to use the same tag more than one time. You can use a simple tag such as 'a' or 'A' to mark the beginning of a section, in this case, *Section A*. This allows the user to jump easily to the start of the section.

List fields

If a series of fields have the same definition, you can list them together in the FIELDS section. For example:

```
FIELDS
 A, B, C : (red, yellow, green)
```

These will all be the same enumerated type, and it is possible to perform assignments between them.

You can list fields together for any type definition, for example:

```
FIELDS
 A, B, C : STRING[10]
```

3.2.1 Field types

Blaise provides a large number of field types. They cover most situations you may encounter in daily practice. You can easily prepare and see Blaise field types in the data model `types.bla` in `\Doc\Chapter3` of the Blaise system folder. Types can be defined at the field they are attached to, in a TYPE section, or in a type library.

String type

A string field accepts any text as value, provided the length of the text does not exceed the specified maximum length. An example:

```
Name "What is your name? " : STRING[30]
```

The reserved word STRING indicates that text is expected. The length for this field is specified between square brackets (in the example above, it is 30 characters). The maximum length is 32,768 characters. For compatibility and data space reasons, STRING, without a length qualification, remains STRING[255].

Depending on Mode Library settings and the length of the string declaration, the data entry cell for a string can be shorter than the string (in which case the cell will scroll, if necessary). The data entry cell can be made wider (this may mean only one column per form pane), or the cell can take more than one space vertically.

You can test the contents of, or make an assignment to, a string field using single quote marks:

```
IF SectionName = 'Household' THEN
  Label1  := 'Household Roster'
ENDIF
```

It is common to test or assign values between string fields:

```
IF Name1 <> Name2 THEN
   Name1 := Name2
ENDIF
```

To test for an empty string you can use either the word EMPTY or two single quote marks with no space between them.

```
IF Name = EMPTY THEN...

IF Name = '' THEN
```

A wide variety of functions that operate on strings or give strings as a result are available. See the *Reference Manual* for a complete list of string functions. For very long text responses, see the following section, *OPEN type*.

You can restrict the characters the user can enter, to valid characters and formats using an edit mask. With an edit mask, if the user attempts to enter a character that is not valid, the character is not accepted. See section *6.6 Data Model Properties*.

OPEN type

For long, open-ended text responses of variable length you can use the OPEN type. OPEN type fields are useful when the interviewer must record verbatim answers from respondents. For example, the survey may ask for a description of a particular injury, or for the respondent to recount an incident in the past. Often, a subject matter specialist will later apply a code to the description.

When the interviewer arrives at an OPEN type field and starts typing, a text entry window opens. This is where the answer is recorded and later reviewed. To review open-ended responses, press the Insert key at the field to open the window. To leave an edit window, press Alt-S (or another shortcut key, as defined in the configuration file).

An example (`opentype.bla` in `\Doc\Chapter3` under the Blaise system folder):

```
DATAMODEL OpenType "Describes an open type question."
  FIELDS
    Injury "Have you ever had an injury? " : (yes, no)
    Describe "Please describe the injury." : OPEN, EMPTY
  RULES
    Injury
    IF Injury = yes THEN
      Describe
    ENDIF
ENDMODEL
```

String fields should be used for short open-ended responses. An advantage of OPEN fields over string fields is that the response is typed in a separate popup window and hence is easier to read. A long string question is recorded in a field in the FormPane and you have to scroll through long responses. OPEN fields should not be confused with remarks, which may be associated with any field. In a Blaise data set, OPEN answers are part of the main data file. In ASCII readout, OPEN answers are put into a separate file with the extension `.opn`.

The maximum size of one OPEN field response is several hundred lines. In a RULES section, you only refer to an OPEN field with an ASK, SHOW, or KEEP instruction. An OPEN type field can be used in an assignment or an expression. You can refer to the text of the OPEN type field as a variable text fill in the text of another field with the ^ symbol.

You can manipulate the value of an OPEN field using Manipula.

Integer type

Integer fields can be defined in several ways:

```
Age1 "What is your age?" "Age of respondent" : 0..120

Age2 "What is your age?" "Age of respondent" : INTEGER[3]

Age3 "What is your age?" "Age of respondent" : INTEGER
{Use this definition only for a LOCAL}
```

The type of the first field, *Age1*, is a strict range. It contains a lower and upper bound. All specified values must be within these bounds. *Age1* must at least be zero and at most 120.

The field *Age2* in the second line is a more general INTEGER type. The value range is limited by the size of the field (3 characters). Values must be within the specified number of characters (digits and sign). Hence, the values of *Age2* must be in the range from -99 to 999.

The field *Age3* will usually accept any integer up to 18 positions. The general advice is to use strict ranges. This is equivalent to a powerful univariate edit. Users will not be able to exceed defined bounds. This is desirable, but make sure the bounds are correctly chosen. For example, if you define *Age* from 1 to 120, you will not be able to record the age of a newborn.

See the section on *Answer attributes* for a discussion of the ASCII representation of the values REFUSAL and DONTKNOW for certain integers.

Decimal or real type

To define a field that accepts decimal numbers, you can use the following type declarations:

```
Ticket1 "How much did you pay for your train ticket?" : 0.00..10.00

Ticket2 "How much did you pay for your train ticket?" : REAL[5, 2]

Ticket3 "How much did you pay for your train ticket?" : REAL[5]
```

The first declaration, *Ticket1,* is the strictest. Values must be between the lower and upper bound and will be displayed on the screen with the number of decimals that is used in the range specification. If you enter a number with more decimals, the value will be rounded to the specified number of decimals. This also applies to results of computations. If the number of decimals of the lower bound of the definition is not equal to that of the upper bound, the maximum of the two is used.

The second format, *Ticket2,* defines a total field width and a fixed number of decimals. For example, *Ticket2* will accept values in the range from -9.99 to 99.99.

The third format, *Ticket3,* only defines the field width. All digits, including a possible minus sign and a decimal point, must fit into the field. For example, the field *Ticket3* will accept real values in the range from -9999 to 99999.

Testing and assignment of numeric fields is straightforward:

```
IF Age < 15 THEN

Age  := 10

Average  := Total/Number
```

Many functions take numeric fields as arguments or give numeric fields as results. Some of these are specifically for integers or real numbers. See the *Reference Manual* for a full list of numeric functions.

Enumerated type or precode

An enumerated field can take as a value one of the items in a list of items. The item is known as a category identifier. A simple example is:

```
Gender  "What is your gender?" : (Male, Female)
```

The list of possible values consists of a number of names separated by commas and enclosed in parentheses. In this example, the list has two category identifiers: *Male* and *Female*.

Each category identifier must follow the conventions for identifiers. You may attach a code number and a text to each category identifier in the list. A code number must be enclosed in parentheses and the text must be enclosed in double quotes. An example:

```
Activity "What is your main activity?" :
         (School (1)  "Going to school",
          Working (2) "Working",
          HousKeep (5) "Housekeeping",
          Other (7) "Something else")
```

If specified, the category texts will be displayed to the user in the info pane; if not, the category names will appear instead. By default the chosen type name will be displayed in the FormPane of the Data Entry Program.

To evaluate or make an assignment with an enumerated field, use the category identifier directly. To determine if a subset of names is in the enumerated field, use the IN notation. Careful consideration of category identifiers means highly readable programming code for the developer:

```
IF Activity = School THEN
  Activity  := Housekeep
ENDIF

IF Activity IN [School, Working] THEN...
```

The ORDINAL (ORD) function is the only function that takes an enumerated field as an argument. It returns the number of the response in the enumerated field. Following is an example involving two differently defined but related enumerated fields that are used to collect time unit information.

```
FIELDS
  AmountOfTime "How much time did you usually spend on the phone at
              work?" : 0..120
  TimeUnit1 "First time unit" : (minutes, hours)
  TimeUnit2 "Second time unit": (hours, day)
```

In the RULES section, the ORD function could be used:

```
RULES
  AmountOfTime
  TimeUnit1
  TimeUnit2
  CHECK
    ORD(TimeUnit1) < ORD(TimeUnit2)
```

The RULES section is covered in the text to follow.

See the section on *Answer attributes* for a discussion of the ASCII representation of the values REFUSAL and DONTKNOW for enumerated fields with eight or nine options.

Type compatibility for enumerated fields

To assign values between enumerated fields, they must have the same type.

Consider the following two situations:

```
{Situation 1}
FIELDS
  A : (Red, Yellow, Green)
  B : (Red, Yellow, Green)
RULES
  A  := B

{Situation 2}
TYPE
  TrafficLight = (Red, Yellow, Green)
FIELDS
  A : TrafficLight
  B : TrafficLight
RULES
  A := B
```

In Situation 1 above, the computation will not pass the syntax check because Blaise will not recognise fields *A* and *B* as having the same type. In Situation 2, it will.

Set type

To allow a respondent to choose more than one item from a list of answers, use a SET field. A SET field may also be known as a multiple precode or code-all-that-apply. Consider the following example:

```
Travel "How do you travel to your work?" : SET [3] OF
       (NoTravel "Do not travel, work at home",
        PubTrans "Public bus, tram or metro",
        Train    "Train",
        Car      "Car or motor cycle",
        Bicycle  "Bicycle",
        Walk     "Walk",
        Other    "Other means of transport")
```

The format is the same as that of an enumerated field with the addition of the reserved words SET [3] OF. This indicates that, at most, three different category values can be selected. By specifying SET OF without a number between brackets, you allow all items to be picked from the list.

If you need to refer to a specific element of a SET field, then use the [ ] notation. For example:

```
IF Travel[1] = Walk THEN...

IF Travel[i] = Walk THEN...
```

Testing for category name in a SET field is done differently than for an enumerated field. The IN notation is used, but the order is reversed. To test for several item categories at once use OR. For example (where *Travel* is the field name and *Train* and *Car* are categories):

```
IF Train IN Travel THEN...

IF (Train IN Travel) OR
   (Car IN Travel) THEN...
```

The CARDINAL function is the only function with a SET field as an argument. It returns the number of responses chosen for a field.

Stored values of enumerated and set fields

The values in enumerated fields and SET fields are stored as code numbers. The first item in the list is assigned code 1, the second gets code 2, and so on. You can specify your own code numbers to overrule this coding scheme. You do that by adding numbers in parentheses between category names and category texts. Consider the following example:

```
Travel "How do you travel to your work?" : SET [3] OF
       (NoTravel (0) "Do not travel, work at home"
        PubTrans     "Public bus, tram or metro",
        Train        "Train",
        Car      (4) "Car or motorcycle",
        Bicycle      "Bicycle",
        Walk         "Walk",
        Other    (9) "Other means of transport")
```

This means that *NoTravel* is coded as 0. Subsequent values are coded in ascending order until a new code is encountered. Here *PubTrans* is coded as 1 and *Train* as 2. The codes of the other values are 4 for *Car*, 5 for *Bicycle*, 6 for *Walk*, and 9 for *Other*.

If you have more than 9 choices in a SET field you should consider starting the first one with code 10. The reason is that the key sequence 113 may be evaluated as 1-13 or 11-3. You can avoid this confusion if all values in the set definition have the same number of digits.

It is not possible to define non-ascending numbers.

Type compatibility for SET fields

You can compute values from one SET field to another, if the same type defines them. This is shown in the following example in `setcomp.bla` (also in `\Doc\Chapter3`):

```
DATAMODEL SetComp "Data model to test computations in set questions."

TYPE
  TColours = (red, yellow, blue)

FIELDS
  MyColours "Enter my favorite colours.": SET OF TColours
  YourColours "Enter your favorite colours.": SET [2] OF TColours
 . . .

RULES
  MyColours
  YourColours := MyColours
```

The assignment *YourColours := MyColours* works even though *YourColours* takes only two values and *MyColours* can take up to three. If three choices are entered in the field *MyColours*, the first two responses will be computed into *YourColours*.

You can also do a direct computation into a SET field with the following syntax:

```
FIELDS
  HisColours : SET [2] OF TColours

    . . .

RULES
  HisColours := [red, blue]

{
  or for example

 HisColours := [red]
}
```

You can refer to a specific element of a SET field with the array notation using square brackets. In the following example, the enumerated field *MyFavouriteColour* gets the value of the first element of the SET field *MyColours.*

```
FIELDS
  MyFavoriteColour "My favorite colour is the first entry of MyColours."
                   : TColours

RULES
  MyFavoriteColour := MyColours[1]
```

The assignment from a SET field to an enumerated field is done through specific reference to the first element of the field *MyColours.*

In a variable text fill, you can also refer directly to the specific elements of a SET field as shown in the following example:

```
FIELDS
    ShowFill "This shows the values of the elements of MyColours.
             @/
             @/MyColours[1] = ^MyColours[1].
             @/MyColours[2] = ^MyColours[2].
             @/MyColours[3] = ^MyColours[3]."
             : STRING[1], EMPTY
```

Dynamic text in types

You can have dynamic type texts for enumerated or set types. This is accomplished by using text fills in the category texts. This is used in a variety of ways, from simple tailoring of a few words in a text, to the construction of fully

dynamic content for all type responses, in order to implement variable lists and randomly ordered response choices. Here is an example:

```
TYPE
 THHList  = (Person1 "^HHPerson[1]",
             Person2 "^HHPerson[2]",
             Person3 "^HHPerson[3]",
             Person4 "^HHPerson[4]",
             Person5 "^HHPerson[5]"
            )
AUXFIELDS
  HHPerson : ARRAY[1..5] OF STRING[20]
FIELDS
  Person: ARRAY[1..5] OF BPerson
  MedHelp "Who needs Medicare help?" : THHlist
RULES
FOR I:= 1 TO HHSize DO
  Person[I]
  IF Person[I].Age > 59 THEN
    ExitSwitch := 0
    FOR J := 1 TO 10 DO
      IF HHPerson[J] = EMPTY AND ExitSwitch = 0 THEN
        HHPerson[J]:=Person[I].Name
        ExitSwitch := 1
      ENDIF
    ENDDO
  ENDIF
ENDDO
MedHelp
```

Reserving extra space for enumerated and set fields

All the values for an enumerated or set field are not always identified before an instrument is fielded. In these cases you can reserve extra space for added future options.

To reserve space create an enumeration or set field. For each category that will be used as a place holder for future choices, make the category texts a null value (two double quotes).

The syntax to reserve space for both enumerated and set fields are similar. The syntax for the type of a set field is shown in the following code example:

```
TYPE

  TTravel =
     (NoTravel (01) "Do not travel, work at home",
      PubTrans (02) "Public bus, tram or metro",
      Train    (03) ,
      Car      (04) "Car or motor cycle",
      Bicycle  (05) "Bicycle",
      Walk     (06) ,
      Future1  (07) "",
      Future2  (08) "",
      Future3  (09) "",
      Other    (97) "Other means of transportation")
```

When reserving extra space it is not always desirable to display the category identifiers that will be used for future choices. In order to hide those place holders, the enumeration or set must be defined as a user type. See the example code above. After creating the type, select the *Project/Datamodel properties* menu item, the type tab and then from the tree, the user type you created. Check *Hide empty values*.

If need be, you can also hide the category codes. To accomplish this, toggle the field *Show Code Numbers* on the info panes layout tab in the mode library.

Date type

A date field is a special field type that only accepts dates as values. For example:

```
Birth "What is your date of birth?" : DATETYPE
```

For data entry, Blaise uses the date format that is specified under Windows® regional settings (the short date format). The date separator can be a space or the separator according to the Windows® regional settings (for example, /). The valid range is from *1/1/1* to *31/12/9999*.

For developers, a date field can use a wide variety of date functions. This eliminates the need to program tricky date calculations. For example, the following will give the current date according to the computer, the number of the week in the year, the number of the day of the week, the current year of the field *BeginDate*, and the number of days since 1-1-1:

```
DateNow := SYSDATE

NumWeek := WEEK(BeginDate)

NumWeekDay := WEEKDAY(BeginDate)

NumYear := YEAR(BeginDate)

JulianDate := JULIAN(BeginDate)
```

To calculate the number of days in the current year, where the field *NewYearsEve* gives the date of New Year's Eve of the current year (calculated elsewhere):

```
DaysSoFar  := TodaysDate - NewYearsEve
```

To add a number of days to a date, you can use any of the following first three forms:

```
NewDate  := BeginDate + 10

NewDate  := BeginDate + (10)

NewDate  := BeginDate + (0, 0, 10)

NewDate  := BeginDate + (1, 1, 1)

NewDate  := BeginDate + (NumYears, NumMonths, NumDays)

NewDate  := BeginDate - (NumYears, NumMonths, NumDays)
```

In the first line, since there is just one integer with no parentheses, Blaise will assume the 10 to be a number of days. To add just a number of days you could also use the abbreviated *DeltaDate* notation, as shown in the second line, or the full *DeltaDate* notation in the third line. In order to add years or months to the date, you must use the full *DeltaDate* notation as shown in the fourth and fifth lines. You can subtract from a date as shown in the sixth line. The entries in the *DeltaDate* notation can be integers or expressions.

For a ready-made demonstration of time and date functions, you can prepare the data model `timedate.bla` in `\Doc\Chapter3` of the Blaise system folder. Refer to the *Reference Manual* for a full listing of date functions, When using the SYSDATE or STARTDATE functions, you must make sure all computers have the date correctly set.

Blaise is year 2000 compliant. It stores dates with all four digits of the year. The default setting in the Data Entry Program (DEP) configuration file is to enter all

four digits of the year, but you can change this to allow the user to enter only two digits for the year. (See Chapter 6 for information on the DEP configuration file.)

Time type

Another special field type is the time field, which stores time values. Here is an example:

```
TimeTravel "At what time do you start work?" : TIMETYPE
```

Acceptable time values have to consist of three elements: hours, minutes, and seconds. Seconds will only be shown if they are non-zero or if specified in the regional settings of Windows®. All three have to be specified as integer values, separated by a colon (:).

Examples of accepted inputs are:

```
8:12:3        8:12:03        08:12:0300        8:12
```

The latter example shows that you may leave out the seconds. You can also leave out minutes. In the example above, 8:12:3 is recorded as 8:12:03, not as 8:12:30.

The user can enter a time using either the *am/pm* notation or the 24-hour clock. This depends on how the time setting on the computer is set up. The default display setting for time fields is for hours and minutes, even though the seconds (and hundredths of seconds) are recorded internally.

A large number of time functions can be used with time fields. These are very similar to the date functions documented above. You can prepare the example data model `timedate.bla` in `\Doc\Chapter3` for a demonstration. Also refer to the *Reference Manual* for a full list of time functions. When using the functions SYSTIME or STARTTIME, it is imperative that all computers have the correct time set.

The function STARTTIME gives the time when the current form was brought onto the screen. This time will stay constant until the form is closed. When the form is brought back onto the screen, the STARTTIME field will be recomputed.

The function SYSTIME will return the computer time at the instant the rules invoke it. Usually you want to use SYSTIME to capture a time stamp, for example, to see how long it takes to progress through sections of an interview. In this case, you

must embed the SYSTIME function in an IF condition so that it is calculated only one time:

```
TimeStamp.KEEP
IF TimeStamp = EMPTY THEN
  TimeStamp  := SYSTIME
ENDIF
```

The first instruction, *TimeStamp*.KEEP, is necessary to put the field on the route. If you omit this instruction, *TimeStamp* will always be empty in the condition.

Field methods such as ASK, KEEP, and SHOW are covered in the text to follow.

Classification type

To perform hierarchical coding, a classification type is needed. Hierarchical coding and the classification type are discussed in Chapter 5.

Arrayed fields

An arrayed field defines a series of fields of the same type. Arrayed fields are useful for storing characteristics of a series of objects, like the gender of all the members of a household. For example:

```
Gender "Are you male or female?" : ARRAY [1..10] OF (Male, Female)
```

This represents a series of 10 fields, with the names *Gender[1], Gender[2], ..., Gender[10]*. Arrays are particularly powerful if the basic field type is a block or a table. For example, you can have an arrayed block within a table, then array the table itself. This offers the possibility of implementing multilevel rostering (hierarchical) and relational data models. For more information about arrayed blocks and tables, see Chapter 4.

3.2.2 TYPE section

We described the basic field types of the Blaise language in the section above. There, all types were defined at the field to which they were attached. With the TYPE section, you can define reusable type definitions (or just types), then use them in later sections. The general scheme for defining a user type is:

```
TypeName = FieldType
```

TypeName is an identifier. It follows the rules for identifiers. *FieldType* is one of the field types described above. This is illustrated in the following example:

Suppose you have a survey that asks about current commuter behaviour and commuter behaviour one year ago. There are two questions about transport to work. One refers to the current situation and the other to last year. These two questions have different question texts but the same type defined in a type section:

```
TYPE
 TTravel = SET [3] OF
          (NoTravel "Do not travel, work at home",
           PubTrans "Public bus, tram or metro",
           Train    "Train",
           Car      "Car or motor cycle",
           Bicycle  "Bicycle",
           Walk     "Walk",
           Other    "Other means of transport")
FIELDS
 TravelNow  "How do you travel to your work at the moment?" : TTravel
 TravelThen "How did you travel to your work one year ago?" : TTravel
RULES
 TravelNow
 TravelThen
```

TTravel is a reusable type that can be used to define fields. The two fields *TravelNow* and *TravelThen* are defined by referring to *TTravel*.

A good programming practice is to use the convention that the names of user-defined types start with a capital *T*. This makes it easy to distinguish type names from other identifiers.

Advantages of TYPE sections

Defining types in the TYPE section has a number of advantages:

- It reduces the amount of coding you have to do.
- It avoids the risk of recording differences in responses that should be exactly the same.
- Maintaining existing data models becomes much easier. If you decide to change a field type that is used several times in your data model, you only have to make one change.
- The use of reusable types allows a clearer view of the data model.

- If you want to directly compute the value of one enumerated field into another, the common use of the same type is an absolute necessity.
- Data entry masks are defined in terms of types, for example Tdollar for a currency amount

For these reasons, we strongly encourage you to use a TYPE section.

In defining types you may use other user-defined types. In the following example, *TTravel* is a type of the previously defined type *TTransport*:

```
TYPE

TTransport = (NoTravel "Do not travel, work at home",
  PubTrans "Public bus, tram or metro",
  Train    "Train",
  Car      "Car",
  Bicycle  "Bicycle"
  Walk     "Walk",
  Other    "Other means of transport")

  TTravel = SET [3] OF TTransport
```

Types from the TYPE section can be used to define fields, auxfields, additional user-defined types, set types, and block parameters (covered in Chapter 4).

A *Block* may be considered a *FieldType*. This is so important for efficient and wise programming in Blaise that all of Chapter 4 is devoted to this and related concepts.

Type libraries

Survey organisations will find it profitable to standardise the types that are used in two or more surveys. This will make programming quicker and make it easier to compare survey results. Interviewers and data editors will also appreciate the common standards between surveys. Types can be shared by many applications in two different ways. They can be incorporated with an INCLUDE statement or they can be stored in a pre-compiled type library.

A TYPE section can be held in another text file and incorporated in a data model with the INCLUDE statement. For example:

```
INCLUDE "Mylib.lib"
```

The type library is prepared just like a data model except that the first key word is LIBRARY. A very simple type library:

```
LIBRARY MyLib
  TYPE
    TYesNo = (Yes, No)
    TMarStat = (Single, Married, Divorced)
ENDLIBRARY.
```

To use the type library from a data model, name the type library in a LIBRARIES section. Then, for any field in the data model, refer to any type in the usual manner. For example:

```
DATAMODEL UsesLib
  LIBRARIES MyLib
  FIELDS
    WillYou : TYesNo
    Married : TMarStat
ENDMODEL
```

You can refer to the file name including path, if necessary. For example:

```
LIBRARIES MyLib'\BlaiseLib\MyLib'
```

If the library is held in the main development directory, the path name is unnecessary.

```
LIBRARIES MyLib
```

Only elementary types can be defined in a type library. SET, ARRAY, and derived types are not allowed in a TYPE library. These can be defined in a TYPE section outside of the library.

In large organisations, the use of several type libraries may make it easier for departments to maintain their types separately. You can use several type libraries in a data model. For example:

```
DATAMODEL ManyTypes
  LIBRARIES
    SocialTypes, EconTypes
```

If there are duplicate type names between type libraries, the latest definition will be used. You can refer to a type in a specific library using the dot notation. In this

example, the field *Married* is specifically defined with reference to the *SocialTypes* library:

```
DATAMODEL ManyTypes
  LIBRARIES
    SocialTypes, EconType
  FIELDS
    WillYou : TYesNo
    Married : SocialTypes.TMarStat
ENDMODEL
```

Another way to guard against duplicate type names between different departments is to define one type library from several different ASCII files. For example:

```
LIBRARY AllTypes
  INCLUDE "SOCIAL.LIB"
  INCLUDE "ECON.LIB"
END.
```

If there are duplicate type names in this latter example, they will be caught during preparation of the library. The first lines of the main data model file would look like this:

```
DATAMODEL ManyTypes
LIBRARIES
    AllTypes
```

You should avoid having someone change a type definition without telling others. If this happens, instruments may not prepare. There may be database incompatibilities, or you may have an instrument that works in an unexpected way.

Not all types have to appear in the type library. If there are types used by just one survey, these can be mentioned separately in the survey data model in a type section. The extension of the type library is `.bli`.

Where to define types

Use the following rules to determine where to define types. They are not required, but they do represent good programming practice:

- If the type is used one time in one survey, define the type at the field in the FIELDS section or in a TYPE section.
- If the type is used twice or more in one survey, define the type in the TYPE section of the data model.
- If the type is used in two or more surveys, define it in a type library or in an INCLUDE file of types.
- If the type is a dynamic type, it has to be declared after the local, auxfield, or field that defines the dynamic type text.
- If the type is a dynamic type using several or many array elements, define the type in the block where it is used. This is for performance reasons as explained in Chapter 4.

You can use all of these methods in the same data model. This could be done in a data model of any size.

It is easier to include a text file of a TYPE section than to prepare a type library. However, a type library protects against changes and, in extremely large instruments, preparation of the instrument may be faster with a pre-compiled type library.

3.2.3 Answer attributes

We complete the description of fields by describing the special answer attributes you can assign to them. These are available in the Data Entry Program by using a special key. The following special attributes are available:

- DONTKNOW or DK: *Don't know* response is allowed.
- NODONTKNOW or NODK: *Don't know* response is not allowed; this is the default.
- REFUSAL or RF: *Refusal* response is allowed.
- NOREFUSAL or NORF: *Refusal* response is not allowed; this is the default.
- EMPTY: The field can be left empty, even if on route.
- NOEMPTY: A value must be entered or computed; this is the default.

Internally, Blaise stores DONTKNOW and REFUSAL as statuses associated with a field, not as special numeric entries in a field.

You can specify special attributes after the answer definition (see the text to follow about setting attributes at the block or data model level). Use commas to separate attributes from the answer definition and from each other. The scheme for the special attributes is:

```
FieldName : FieldType, SpecAttr1, SpecAttr2, ...
```

SpecAttr1, *SpecAttr2*, ... denote the special attributes. The default attributes are NODONTKNOW, NOREFUSAL, and NOEMPTY. The following examples illustrate how these attributes can be used:

```
Gender      "What is your Gender?"
            : (Male, Female), NODONTKNOW, NOREFUSAL

Distance "What is the distance to your work?" : 0..300, EMPTY
```

In the above example, the question *Gender* must really be answered; the answers 'don't know' or 'refusal' are not accepted. The question *Distance* may be left unanswered.

The EMPTY attribute should be used judiciously. You do not want interviewers to be able to avoid questions that must be asked. If the interviewer has navigated backward and presses the End key on the keyboard, she will skip past a field on the route that has the EMPTY attribute, even if it has not yet been asked.

Fields with the SHOW or KEEP methods must get values through computation. These methods are covered in the text to follow. Make sure that they have a value definition large enough to accept any possible computation that can happen in the instrument. If a field is assigned a value outside its range, the user will get a message about an imputation error and will not be able to continue. This is demonstrated by the data model `testcomp.bla` in `\Doc\Chapter3`.

A field which has the status REFUSAL or DONTKNOW or holds the EMPTY value will be evaluated as a zero when referring to its value in an IF condition. In these cases, you must be very explicit in what you want to happen.

For example:

```
FIELDS
  ...
  TotalDist "What is the total distance to your work?"
            : 0.0..300.0, REFUSAL, DONTKNOW, EMPTY
  MethodOfTransport "What is the primary method of transport?" :
                    TTransport, RF

RULES
  IF TotalDist <> EMPTY THEN
    MethodOfTransport
  ENDIF
```

The field *MethodOfTransport* is asked if *TotalDist* has a positive value or an answer of DONTKNOW or REFUSAL.

The IF condition above could have been written as either of the following, depending on the needs of the survey:

```
IF (TotalDist > 0) OR (TotalDist = DK) THEN

IF TotalDist = RESPONSE THEN
```

While the value of DONTKNOW or REFUSAL is evaluated as zero internally, the ASCII representation is different. In the ASCII data set, the value for REFUSAL is all 9s with the last digit of 8. The ASCII value for DONTKNOW is all 9s.

For example, for a field with range 1..200, the ASCII representation of refusal will be 998, while the representation for DONTKNOW will be 999. If the value of REFUSAL or DONTKNOW in ASCII would be a valid value within the defined range, then the number of characters taken for the field in ASCII will increase by 1. Thus, in ASCII a field with a defined range of INTEGER[3] will represent REFUSAL as 9998 and DONTKNOW as 9999.

The same thing will happen with enumerated questions with eight or nine choices. This will be transparent to the user. If data are read out of and then into Blaise, all REFUSAL and DONTKNOW statuses will be preserved. Cameleon will correctly specify the ASCII representation, taking into consideration values for REFUSAL and DONTKNOW where applicable.

See the data model `statuses.bla` in `\Doc\Chapter3` for a thorough review of statuses and their effect on the ASCII representation.

The ASCII representation of REFUSAL and DONTKNOW may be problematic when importing the ASCII data into another package. There are three solutions:

- Run the ASCII data through a Manipula program you create to convert the REFUSAL and DONTKNOW values into something the other package is expecting.
- Use the other package to make the translation.
- Define your own REFUSAL and DONTKNOW values in the applicable field definitions.

For example, if -1 is the value expected by the other package for nonresponse, you can define a numeric type as follows:

```
TYPE
  Sev9sM = -1..9999997  {seven digits from minus 1}

FIELDS
  Money : "How much money do you have with you?" :  Sev9sM
```

To avoid having an ASCII data set expand to handle values for REFUSAL and DONTKNOW, do not use ranges such as 1..999 or INTEGER[3]. Some survey organisations set up special types to handle such integers. They define integers as reusable types in the TYPE section with a 7 as a last digit. For a three-digit integer field, this will preclude using 998 or 999 as valid answers. If these are potentially valid, you probably need an extra digit anyway. See the type *Sev9sM* in the example above.

Attributes may be given at the block or instrument level in a settings paragraph with the key word ATTRIBUTES. For example:

```
BLOCK Attrib
  SETTINGS
    ATTRIBUTES = REFUSAL, DONTKNOW,
  FIELDS
    . . .
```

Settings are applied to every field in the block. You can override the block settings at a field by applying converse settings at the field. Block attributes reduce programming code and make it more readable.

You can also set global attributes at the data model level. This is done similarly, but the key word SETTINGS is omitted.

For example:

```
ATTRIBUTES = REFUSAL, DONTKNOW
```

3.2.4 Enhancing texts

Texts and descriptions are used in many places throughout the Blaise language. For the interviewer you can enhance text in ways that clarify the question text or instructions. You can use different fonts and font sizes, colours, bold, underline, or italics to highlight certain words; spaces and tabs to separate words of text; and linefeeds to separate lines of text. Fields, auxfields, or locals can be variable fills within the text. Cue words can be defined that mean something special to the interviewer.

Your organisation should state and enforce text-enhancement conventions. Conventions make it very easy for an interviewer to switch between applications. This would make the interviewers happy and avoid a lot of reformatting of field texts when the complaints pour in or when the organisation decides to standardise after the fact.

You could have different text enhancement conventions for each of the following:

- Fills
- Name of respondent
- Question text
- Information text
- Designation of a hard error
- Designation of a soft error
- Difficult concepts.

You may have conventions for text spacing, such as:

- Question text appears by itself
- One line space between question text and instruction text.

Spacing

When Blaise displays text in the InfoPane, multiple, leading, and trailing spaces in the source code are ignored. If text in the source code file continues on the next line, one space is inserted.

Examples of texts are:

```
"What is your name?"

"You are too young to be married!"

"What is the distance (in km) to your work?"
```

On the InfoPane the latter example will be converted to

```
What is the distance (in km) to your work?
```

as all but one space between words will be ignored.

The Data Entry Program (DEP) will take care of wrapping the text if necessary, so that it will always be readable by the user. A setting in the mode library program allows you to set text margins in the info pane. See Chapter 6, Section 6.5.4, *Mode library file: Layout.*

You can insert special layout commands to control the screen layout. For example, a new line command is generated by inserting @/ in the text. In the following example, we have included this command:

```
"What is your address? @/Please enter street and number:"
```

When this text is displayed on the screen, it will look like this:

```
What is your address?
Please enter street and number:
```

Hard space

Usually spaces in text are skipped. If you want a space in your text that will not be skipped, hold the Ctrl key and type a period (Ctrl-.). Typing the following in the source code:

```
"What is your address? @/·····Please enter street and number:"
```

will appear on the screen as:

```
What is your address?
     Please enter street and number:
```

if a non-proportional font is used. If you use a proportional font, it is best to use tab stops to space text (see the text to follow).

Font, font size, colour, bold, and underline

You can use different fonts, font sizes, colours, bold, underline, or italics to improve the readability of texts on the screen. Adding enhancements to text is a two-step process. First, you assign meaning to special enhancement codes @A to @Z (if you do not want to use the defaults). This is done in the mode library file. For example, you could assign the colour code for red text to the enhancement marker @R and you could assign codes for underscore and bold text to the markers @U and @B.

Second, insert an enhancement marker in the question text. Commands may be nested. If @B stands for bold text, and @G for green, the following text:

```
"@GWhat is your address?@/@/.....@BPlease enter street and number:"
```

will be displayed in different enhancements. The part 'What is your address?' will be green and 'Please enter street and number' will be in green and bold. In other words, the text enhancements are additive.

Repeating the command switches back to the previous enhancement. If @B is the colour command for bold text, the initial colour of the text

```
"@GWhat is your @Baddress@B?@/Please enter @Bstreet@B and @Bnumber@B:"
```

is green. The words *address*, *street*, and *number* will be green and bold.

You do not have to switch off enhancements at the end of the text. This is done automatically. If you want the whole text in a specific enhancement, one command at the beginning is sufficient.

The introduction of fonts increases screen design flexibility. With proportional fonts, control of font sizes, and judicious use of bold, italics, and underscore for interviewer instructions, it is possible to place a great deal of information in the question text part of the DEP window. An important ramification of this is that the page part of the screen (the FormPane) can also contain more information. A full description of fonts and how to use them is in Chapter 6.

An advantage of having text-enhancement assignments stored externally is apparent in multi-platform environments. For example, you may have an application running on in-office CATI with one screen resolution and the same application on laptops with a different resolution. By using two different versions of the external configuration file, you can use the same instrument and have appropriate displays in both environments. Screen resolution and font size choice interact to determine how much text can fit in the InfoPane (question text part) of the interviewing instrument. See Chapter 6 for details.

Characters ^, @, "

The characters ^, @, and " have a special meaning in texts. If you want to use those characters as normal symbols, you have to write them twice. For example:

```
"Do you know the characters ^^, @@ and ""?"
```

is displayed as:

```
Do you know the characters ^, @ and "?
```

Tab stops

The use of proportional fonts can increase the readability of text for the user. However, spacing text in a proportional font cannot be done effectively with the special hard space dot character. To space text in a proportional font, use tab stops.

To insert a tab stop in the text, use the symbol @|, either alone or in combination with a letter. For example, the following uses the default tab stops:

```
@|Name @|Address @|Town
```

You can also define tab stops for each letter. The following uses tab stops for the letter T:

```
@T@|Name @|Address @|Town
```

To specify two or more tab stops, use the symbol twice, for example:

```
@T@|Name @|Address @|@|Town
```

The above would place two tab stops in front of *Town.* Chapter 6 describes how to set tab stop definitions for letters in the DEP configuration file.

Variable text fills

To improve the readability of texts, you can use information that comes from other fields in the data model. You can refer to a parameter field, auxfield, or local variable by including its name preceded by the special character ^. A field can be anywhere in the data structure, as long as you specify the correct path to it. A local variable must be in the scope of the current block. If you have a field called *Name* in the same block as the text, then you can use the value of this field in a text in the following way:

```
FIELDS
  Name "What is the name of the head of the household?" : STRING[20]

  Age  "What is the age of ^Name?" : 0..120
```

The DEP will replace *Name* in the text of *Age* by its current value. For example, if the field *Name* contains the value 'Martin Turbo,' the text becomes:

```
What is the age of Martin Turbo?
```

If you refer to a field that is not in the current block or a higher block, you can use a parameter reference (preferred) or the dot notation. You will find more about this in Chapter 4.

Text fills use data model properties formatting, e.g., social security number, telephone number, or a date type.

For an example of formatting of field and edit check text, prepare and view `format.bla` in `\Doc\Chapter3` under the Blaise system folder.

3.3 Auxiliary Fields (Auxfields)

You may want the Data Entry Program to put an entry cell in the FormPane of the screen without storing its value in the data file. This is done with an auxiliary field or an auxfield. An example would be a screen label that is a visual guide to those who are paging through the instrument. Another example is a message. Sometimes you may want to display a message on the screen that contains information that should be read before one continues to the next question. You would also use an auxfield if you want to perform an intermediate calculation involving enumerated types or if you want its value to be re-initialised every time the form is brought into memory, but not every time the rules are invoked.

Do not use auxfields as filter questions to control routing! If you depend on auxfields for routing questions you will get through the questionnaire correctly the first time. However, if you retrieve the form and then store it without filling in the filter questions again, you will lose data!

Auxfields section

The structure of the AUXFIELDS section is identical to that of the FIELDS section. The only difference is how you announce this section: write AUXFIELDS instead of FIELDS.

The following partial model specification contains an example of an AUXFIELDS section:

```
FIELDS
  Job      "Do you have a paid job?" : (Yes, No)
  Descrip  "Give a short description of your job" :  STRING[40]
  WorkHome "Do you work at home" : (Yes, No)
  Distance "What is the distance to your work?" : 0..300
  Travel "How do you travel to your work?" : SET [3] OF TTravel

AUXFIELDS
  Info    "The next two questions are about how you travel to your work.
            @/ Please try to describe the normal situation.
            @/@/ Press Enter" : STRING[1], EMPTY

RULES
  Job
  IF Job = Yes THEN
    Descrip
    WorkHome
    IF WorkHome <> Yes THEN
      Info
      Distance
      Travel
    ENDIF
  ENDIF
```

The auxfield *Info* displays a message. It will appear on the screen just before the questions *Distance* and *Travel* are asked. Note the answer type of *Info*. By defining it as STRING[1], we allow any symbol to be accepted as answer, and by adding the attribute EMPTY, we allow the auxfield to remain empty. Pressing Enter is sufficient to continue to the next question. The question *Info* is defined in the AUXFIELDS section, so its value will not be stored.

You can mix FIELDS and AUXFIELDS sections. Remember to put the FIELDS key word after the AUXFIELDS section; otherwise you will find that some data are not stored.

```
FIELDS
  FirstName "What is your first name?" : STRING[15]
  SurName "What is your surname (last name)  ^FirstName?" : STRING[20]
AUXFIELDS
  Info "Now we want to know about your job."  : STRING[1], EMPTY
FIELDS
  HaveJob "Do you have a job?" : (yes, no), REFUSAL, NODONTKNOW
```

Auxfields used as a page label

An important use of an auxfield is to place a label in the page of the interviewing screen. This kind of label can be used to announce a section and to give the interviewer a landmark that can be used for page-based navigation. If you use a field tag that is the section letter, the interviewer can jump to the section any time

that it is on the route. In the following example, the interviewer can jump to *A* to get back to section *A* if it has already been filled in and passed.

```
AUXFIELDS
  Label (a) "@W[INTERVIEWER] You are entering section A on Demographic
             data."  : STRING[20], EMPTY
```

In the RULES section, you make a string computation and place the auxfield in the rules above the other fields in the block.

```
RULES
  Label := 'Demographic data'
  Label
  FirstName
{etc.}
```

Rules and computations are covered in the following section.

3.4 Local Variables (Locals)

Another entity that can hold information is the local variable, or local. Its value is never asked or displayed on the FormPane but is always computed. Locals are used as control variables to loop through arrays, and you can use them to store intermediate results in complex computations. Such computations are sometimes necessary to determine the correct route through the fields. Edits may also require computations, and locals are often used in edit messages to display the result of such calculations. See the following example of a local used in the RULES section.

The life span of a local is even shorter than that of an auxfield. Whenever a RULES section is invoked during an interview or data editing, locals are reinitialised. This means that *every time you change a value in a field, the local variable loses its current value*.

Locals section

Locals are declared in a section that starts with the reserved word LOCALS. You can have integer, real, or string variables, but not enumerated types. You can define arrays of local variables:

```
LOCALS
  Total, Ncases : INTEGER

  Average       : REAL

  JobDescr      : STRING[40]

  Names         : ARRAY [1..20] OF STRING[16]

  Weights       : ARRAY [1..10, 1..5] OF REAL
```

Total and *Ncases* are local variables of the same type, so they can be introduced in one statement. They are both integer variables. *Average* is a real variable, and *JobDescr* is a text variable that can hold at most 40 characters. *Names* is an array of 20 text variables *Names[1], Names[2], ... Names[20]*. The final instruction introduces a two-dimensional array. Such a construct can be useful for working with tables (see Chapter 4).

The scope of local variables is limited to the block and sub-blocks in which they are defined. It is not possible to refer to such a variable outside its block and nested sub-blocks. When using locals as control variables for loops, it is required that the local be declared in the block to which the loop belongs.

A good programming practice in Blaise is to declare locals at the lowest block level possible. This allows the block to be more independent of the rest of the code and hence more reusable in this and other applications. Just as importantly, declaring locals at the lowest level possible will help ensure that performance of applications is optimal.

3.5 Summary of Fields, Auxfields, and Locals

The following table summarises the properties of fields, auxiliary fields, and local variables (locals):

Figure 3-1: Summary of fields, auxfields, and locals

	Field	**Auxfield**	**Local**
Value	Asked or computed	Asked or computed	Computed
Storage	In database	Temporary, in memory	Temporary, in memory
Scope	Can be used outside block in which declared	Can be used outside block in which declared	Only within block in which declared (and sub-blocks)
Lifetime	Permanent	Only for current form/case	Only for current form/case
Re-initialised	Never	Every time form is brought into memory	Every time RULES section is processed
Type	All types valid	All types valid	Only REAL, INTEGER, STRING[ ], and ARRAY
Display	Can appear in the FormPane	Can appear in the FormPane	Cannot appear in the FormPane
Methods	Can take ASK, KEEP, and SHOW	Can take ASK, KEEP, and SHOW	Cannot take methods
Text in InfoPane	Associated text can appear in the InfoPane	Associated text can appear in the InfoPane	No associated text
Variable text fill	Can be used as a fill	Can be used as a fill	Can be used as a fill
Effect on block (re) checking if passed to the block	Causes block recheck only if its value changes	Causes block recheck only if its value changes	Causes block recheck only if its value changes

3.6 Rules

The RULES section describes how fields are processed. It starts with the reserved word RULES. There are four types of rules: route instructions, edit checks, computations, and layout instructions. If this section is omitted, all the fields that are specified in the FIELDS section will be processed in the same order they are specified.

3.6.1 Route instructions

Route instructions describe the order of processing for fields and auxfields. Processing the route means deciding which questions will be asked at what time.

Writing down the name of a field or auxfield means processing it. The simplest form of routing instructions is the following:

```
RULES
  Name Gender Age Married
```

or:

```
RULES
  Name
  Gender
  Age
  Married
```

since layout of the source code is free. These route instructions tell the system to process the fields (to ask the questions) in the specified order.

3.6.2 Route field methods

When you write the name of a field or an auxfield in a routing instruction, you instruct Blaise to process the field. This means that the system must apply a *processing method* to the field. Blaise has four methods: CLASSIFY, ASK, SHOW and KEEP. ASK is the default.

You instruct the Data Entry Program (DEP) to apply a method by adding the method name to the field name with the dot notation where the field name is followed by a full stop (period) and the method name. If the field name is *Town*, you may write either *Town.*ASK, *Town.*SHOW, or *Town.*KEEP. Since the default

method of a field is ASK, the two routing instructions *Town* and *Town.*ASK are equivalent.

CLASSIFY

The CLASSIFY routing method puts a classification type field on the route, to show and edit it. See *Chapter 5, Special Topics*, for more information on classification types and the use of CLASSIFY.

ASK

The method ASK shows the field on the screen so the user can enter a value.

SHOW

The method SHOW shows the field and its value on the screen. In the FormPane (page), the user will not be able to land on it and will not have the opportunity to change the value. You typically SHOW a field to give information to the user, as with a label in the page. It is possible to use the mouse to land on a show field, but you can not change its value.

KEEP

The method KEEP means that the field is not shown on the screen. The user will not be aware that this field exists. The value of the field will be stored in the Blaise data file if the field is defined in the FIELDS section, but the KEEP is considered to be at the end of the RULES section.

If you SHOW or KEEP a field, its value must be computed. If a field is computed but otherwise not mentioned in the RULES, the KEEP method will automatically be applied at the point of computation.

If a field is defined in the FIELDS section but is not mentioned in the RULES section, it gets the KEEP method.

Here is an example of the use of the ASK, SHOW, and KEEP methods similar to the `showkeep.bla` found in `\Doc\Chapter3` of the Blaise system folder.

```
DATAMODEL ShowKeep "Demonstrates SHOW, KEEP, and ASK methods."
  FIELDS
    Intro "This is an instrument to show you the effect of SHOW and KEEP
           field methods on the display of fields on the FormPane." :
           STRING[1], EMPTY
    TodayDate "Today's date" : DATETYPE
    TimeStamp "Interview completed" : TIMETYPE
    Name "What is your name" : String[20]
  RULES
    TodayDate.SHOW
    Intro.ASK
    IF TodayDate = EMPTY THEN
      TodayDate  := SYSDATE
    ENDIF
    TimeStamp.KEEP
    IF TimeStamp = EMPTY THEN
      TimeStamp  := SYSTIME
    ENDIF
    Name.ASK
ENDMODEL
```

The question asked is *Name*. *Name*.ASK is equivalent to *Name*. The field *TodayDate* is assigned a value by means of a computation. It is the result of the standard function SYSDATE that reads the system date of your computer. The value of *TodayDate* is shown on the screen.

The value of the field *TimeStamp* is computed but not shown.

The value assigned is the result of the standard function SYSTIME, which reads the system time of your computer. *TimeStamp* is stored in the data set and the user never sees it.

Notice the construct:

```
TimeStamp.KEEP
IF TimeStamp = EMPTY THEN
   TimeSTamp  := SYSTIME
ENDIF
```

It is necessary to put *TimeStamp*.KEEP before the IF condition involving the field *TimeStamp* so that the rules would know of the value of *TimeStamp*. Without the *TimeStamp*.KEEP placed before the IF condition, the IF condition would evaluate *TimeStamp* as EMPTY even if it already had a value.

Preventing return to some questions

You can use the KEEP method to prevent users from returning to some questions after they have been answered. This is called a wall. This might be done for

reasons of confidentiality, as shown in the example keepdemo.bla (under \Doc\Chapter3). In this example, the respondent enters answers directly into the computer for a confidential section on traffic offences. It is felt that the respondent would not answer truthfully if he had to provide answers to the interviewer. A wall will hide the answers from the interviewer once the section is finished. In that way, the respondent can be assured that his answers will be kept confidential from the interviewer. The strategic part of the RULES section of keepdemo.bla is shown in the following example:

```
RULES
  ThankYou.KEEP
  RespondentIntro
  NEWPAGE
  IF ThankYou = EMPTY THEN
    Ticket
    SmallOffence
    MajorOffence

  ELSE
    Ticket.KEEP
    SmallOffence.KEEP
    MajorOffence.KEEP

  ENDIF
  ThankYou
```

The KEEP method is used two ways here. First, it is applied to the field *ThankYou* at the top of the RULES section. In this way, the value of *ThankYou* is known to all of the rest of the fields, even though it does not get a value until nearly the end of the section. If *ThankYou* is empty, then the confidential questions *Ticket*, *SmallOffence*, and *MajorOffence* are asked. Once *ThankYou* is answered, these confidential questions get the KEEP method and can no longer be reviewed or updated.

This method of using a wall can be important for blocks and is demonstrated in the household roster section of Chapter 4.

Forcing a jump back to a previously unanswered question

The KEEP method can also be used to force a jump back to a previously unanswered question. The following example is from jumpback.bla in \Doc\Chapter3, and is illustrative of the method:

```
FIELDS
  StartHere "This is the beginning question." : TContinue
  JumpBackToHere "Since the field QuestionSwitch has answered, you have
                 jumped back to here." : TContinue
  QuestionSwitch "Once you answer this question, you will be forced back
                 to the field JumpBackToHere." : TContinue
. . .

RULES
  StartHere
  QuestionSwitch.KEEP
  IF QuestionSwitch <> EMPTY THEN
    JumpBackToHere
  ENDIF
  QuestionSwitch
```

The field *QuestionSwitch* appears near the top of the rules with the KEEP method so that the succeeding IF condition knows the value of *QuestionSwitch*, even though *QuestionSwitch* will not be answered until later. Once *QuestionSwitch* is answered, the cursor will move backwards in the instrument and land on the field *JumpBackToHere*.

For this to work, it is essential that the field *JumpBackToHere* does not have the EMPTY attribute. When the field *QuestionSwitch* is answered, all the statements in the RULES section are evaluated top to bottom. In this example, the DEP will find that *JumpBackToHere* is now on the route and that it does not have the EMPTY attribute, and thus must be answered. This forces the cursor to move backward.

Note that once the jump back is made, the field *JumpBackToHere* must be answered before the interview can be completed. As always, you should leave the interviewer a way to leave the field, for example, to allow a DONTKNOW or a REFUSAL.

The data model `jumpback.bla` also illustrates using an edit statement to allow an interviewer to jump back to a previous field. You would use this method if the field to which you are jumping back had the EMPTY attribute.

The data model `jumpback.bla` illustrates an important feature in Blaise. All appropriate rules are always re-executed every time a new answer is provided or an existing one is modified. Once the field *QuestionSwitch* receives a value, the DEP knows that the field *JumpBackToHere* needs an answer because the DEP re-evaluated all the rules in the section, top to bottom.

Use of KEEP to aid in some variable text fills

The KEEP method can be applied to a field in the RULES section as many times as you like. In the example data model `filldemo.bla` (in the `\Doc\Chapter3` folder), KEEP is applied at the beginning of the RULES section to three fields.

```
DATAMODEL FillDemo "Data model to demonstrate the use of KEEP for some
                   variable text fills."

FIELDS
  FirstName "First name = ^FirstName,
             Surname = ^SurName
             Age = ^Age
             @/@/What is your first name?" : STRING[30]
  SurName "First name = ^FirstName, Surname = ^Surname, Age = ^Age
           @/@/What is your surname?" : STRING[30]
  Age "First name = ^FirstName, Surname = ^Surname, Age = ^Age
       @/@/What is your age?" : 0..120
RULES
  FirstName.KEEP
  SurName.KEEP
  Age.KEEP
  FirstName
  SurName
  Age
ENDMODEL
```

This is done so that the following set of variable text fills

```
First name = ^FirstName, Surname = ^Surname, Age = ^Age
```

is known at all times for all three fields. In the DEP in this example, you can enter *FirstName*, *Surname*, and *Age* and then move the cursor back to *FirstName*. Even though *FirstName* is in the rules before *Surname* and *Age*, the values of *Surname* and *Age* are displayed for the field *FirstName*. You can experiment and comment out the KEEP statements in the RULES section and see how the variable text fills are handled without them.

This method of showing fills is important in some rostering situations. This is demonstrated in Chapter 4 where several methods of collecting household data are explained.

It is possible to apply ASK, SHOW, and KEEP to blocks. If you apply SHOW or KEEP to a block, then these block-level methods cannot be overridden with field-level methods in the block. If you apply ASK to the block, then SHOW and KEEP at the field level will override the block-level ASK method.

The overuse of KEEP on a block is discouraged. If there is a KEEP on a block, the rules of the block will be processed and the blocks parameters will be administered each time the rules are checked. This may lead to unintended effects if a block is both KEEP and ASK at the same time, as its rules will be processed twice.

Additionally, a KEEP can lead to the unintended effect of leaving ‘orphaned’ data in the database. This can occur when an interviewer is lead down a route and then backs up to take an alternative route. Any fields with a KEEP on the first route will not be emptied when the form is saved.

3.6.3 Conditional rules

Some fields, edit checks, and computations should be processed only in certain circumstances. For example, if you use an interviewing program, certain questions may apply only to certain situations. You do not want to bother respondents with irrelevant questions or skip relevant ones. You need some kind of mechanism to tell the system to ask fields or invoke edit checks or computations only under certain conditions. That mechanism is the IF condition.

IF statement

A conditional rule is described with an IF statement:

```
RULES
  Working
  IF Working = Yes THEN
    Descrip
    Distance
    Transport
  ENDIF
  Income
```

This example only contains route instructions. First, the field *Working* is processed. Then there is a condition specifying that all instructions between THEN and ENDIF (in this case, the fields *Descrip*, *Distance*, and *Transport*) have to be processed only if the value of the field *Working* is *Yes*. For any other value of *Working,* these fields will be skipped.

Finally, the field *Income* will be processed. This field is not included in the part between THEN and ENDIF, and therefore it will always be processed, whatever the result of the condition.

You can also use a conditional rule for checking:

```
RULES
  Name
  Town
  Gender
  MarStat
  Age
  IF Age < 15 THEN
    MarStat = NevMarr
  ENDIF
  Activity
```

The check *MarStat = NevMarr* is only carried out if the field *Age* contains a value less than 15.

You can have a mixture of route, check, and compute instructions between THEN and ENDIF (check and compute instructions are described in detail in the following sections):

```
RULES
  Working
  IF Working = Yes THEN
    Descrip
    Distance
    Time
    Speed  := Distance / Time (Speed > 0.05) AND (Speed < 2.00)
    Transport
  ENDIF
  Income
```

ELSE instruction

Conditional rules can have a more complex structure than those described above. For example, you can use ELSE to specify instructions that will be carried out if the condition is not satisfied:

```
RULES
  Working
  IF Working = Yes THEN
    Descrip Distance Time Transport
    Speed  := Distance / Time  (Speed > 0.05) AND (Speed < 2.00)
    IF Distance > 10 THEN
      Commuter := Yes ELSE Commuter := NO
    ENDIF
  ELSE
    LookWork
    IF LookWork = Yes THEN
      Age < 65
    ENDIF
    Commuter := No
  ENDIF
  Income
```

If the condition is satisfied, the instructions between THEN and ELSE will be carried out. If the condition is not satisfied (*Working* has the value *No*), the instructions between ELSE and ENDIF will be carried out. In both cases, a computation is carried out. The field *Commuter* will always be assigned a value. The field *Income* will always be asked.

In fact, the following is an example of nested conditional rules. There are conditional rules within conditional rules. If *Working* has the value *Yes*, then the system will encounter the conditional rule IF DISTANCE > 10 THEN, and if *Working* has the value *No*, the system will encounter the conditional rules IF LOOKWORK = YES THEN. Blaise allows you to build very complex but also very efficient rule structures.

ELSEIF instruction

In this example, we have an example of the ELSEIF construction. Blaise will evaluate each ELSEIF condition until it finds one that is satisfied. It will process what is contained in that particular part of the ELSEIF construction and then jump to the ENDIF. It will not evaluate succeeding ELSEIF conditions of that particular construction. This makes evaluation very efficient, especially where there are many ELSEIFs together.

```
RULES
  Activity
  IF Activity = School THEN
    SchoolType
  ELSEIF Activity = Working THEN
    Descrip
    Distance
    Travel
  ELSEIF Activity = HousKeep THEN
    LookChildren
    CommunWork
    IF CommunWork = Yes THEN
      Hours
    ENDIF
  ELSEIF Activity = Other THEN
    OthActivity
  ENDIF
  Income
```

! It is good programming practice to pay attention to the layout of the programming code. Standard indentation is particularly important for writing nested structures. Using indentation makes it easier to read and interpret the text and reduces the risk of errors. Particularly, related IFs, ELSEIFs, ELSEs, and ENDIFs should all be lined up in the same column of the text editor. If this is not done, it might be extremely difficult for someone else to understand the code. When IFs and ENDIFs are widely separated by many statements, it is good programming practice to use remarks at the ENDIF to state which IF condition it goes with, as shown in the following example:

```
IF Age > 15 THEN
  . . .
  {Many statements}
  . . .
ENDIF {Age > 15}
```

Specify other choice

A special type of closed question appears often on paper questionnaire forms. It contains a list of options and an alternative if none of the items apply. An example:

```
What is your main activity?
    Going to school ...................... 1
    Working in a paid job ............... 2
    Housekeeping ........................ 3
    Something else. Please specify:
    ---------------------------------------
```

To handle this in Blaise you introduce two fields: one field with four value items and another field designed to store the natural-language specification in case the first field has item 4 as its value. An example:

```
FIELDS
  Activity "What is your main activity?" :
    (School   "Going to school",
     Working  "Working in paid job",
     HousKeep "Housekeeping",
     Other    "Something else")
     OthActiv    "Please specify that other activity" : STRING[80]
RULES
  Activity
  IF Activity = Other THEN
    OthActiv
  ENDIF
```

You can declare placeholders in the enumeration for additional choices. This is described in *Section 3.2.2*.

Field name listed twice

A field name may appear more than once in a conditional rule structure. However, fields that are displayed on the screen (due to the application of the ASK or SHOW method) must always appear in the same order in different branches of conditional rules. They must appear in such a way that they can only be processed once, regardless which route is followed.

The reason for this restriction is a practical one. Screen displays in the DEP are built in advance to ensure a good program performance. This implies that the display order has to be independent of the conditions met in the field. There is no restriction on the use and frequency of the KEEP method and of assignments, because they have no influence on the display. An example follows:

```
RULES
  Activity
  IF Activity = School THEN
    SchoolType
    Distance
    Travel
  ELSEIF Activity = Working THEN
    Descrip
    Distance
    Travel
    Income
  ENDIF
```

There are three possible routes through these fields. If *Activity* has neither the value *School* nor *Working*, no other field is processed. If *Activity* has the value *School*, the fields *SchoolType*, *Distance*, and *Travel* are processed. If *Activity* has the value *Working*, the fields *Descrip*, *Distance*, *Travel,* and *Income* are processed. The two fields *Distance* and *Travel* appear twice in the source code, but once on the screen: once in the *School* branch of the route and once in the *Working* part of the route. The order of the fields is the same in both branches. Remember, all fields are asked unless the SHOW or KEEP method is used.

Fields that are used more than once do not need to use the same method in every branch. For example, it is allowed to ASK a field in one branch and to SHOW it in the other branch. Take a look at the following example:

```
FIELDS
  HomeTown "Where do you live?" : STRING[20]
  SameTown "Do you work in ^HomeTown?" : (Yes, No)
  WorkTown "Where do you work?" : STRING[20]
RULES
  HomeTown
  SameTown
  IF SameTown = Yes THEN
    WorkTown  := HomeTown
    WorkTown.SHOW
  ELSE
    WorkTown.ASK
  ENDIF
```

The question *SameTown* determines whether respondents work in their hometown. If this is the case, respondents will not be asked to specify the town in which they work. Instead, the corresponding field *WorkTown* is computed by giving it the same value as the field *HomeTown*. If respondents work in a different town, they will be asked to give the name of that town.

When you use different methods for the same field in different branches of a conditional rule, you have to keep in mind the possible effects on your data file.

Two possible conditional rules are displayed in the following example:

```
Gender
IF Gender = Female THEN
  Children
ENDIF
```

and

```
Gender
IF Gender = Female THEN
  Children
ELSE
  Children := 0
ENDIF
```

In both examples, the field *Gender* will be filled. In the first one, the field *Children* will always be empty for males. The combination (*Male*, *0*) is not possible. For females, both fields *Gender* and *Children* will always be filled.

In the second example, the effects for the conditional rule are different. Both of the fields *Gender* and *Children* will always be filled. For males, the field *Children* is set to zero.

Error text in IF conditions

If a check is included in a conditional rule, the error message will contain both the condition of the IF instruction and the check specification. If a check is unsuccessful, an error occurs and a message appears on your screen. For the check:

```
IF Age < 15 THEN
  MarStat = NevMarr
ENDIF
```

The message that will appear on your screen is *IF (Age < 15) THEN (MarStat = NevMarr)*.

This message can be improved by adding text. For example, the check:

```
IF Age < 15 "If you are younger than 15" THEN
   MarStat = NevMarr "you cannot be married!"
ENDIF
```

produces the message *If you are younger than 15 you cannot be married!* The IFs and THENs are not displayed. If you want them, you have to include them in the text.

INVOLVING function

Once again we would like to stress the rule that at least one field or auxfield must appear in a check. For a conditional check this means that there should be at least one field in the condition after IF or in the check itself. If you do not have fields in your check, you can force them to be involved by using the INVOLVING function. Consider the following:

```
RULES
  Working
  IF Working = Yes THEN
     Distance
     Time
     Speed  := Distance / Time
     Speed > 0.05 AND Speed < 2.00
  ENDIF
```

Speed is a local variable, so the error cannot be attached to a field. To involve the fields *Distance* and *Time* in the error message, you can use the instruction INVOLVING. This can be achieved with the following:

```
RULES
  Working
  IF Working = Yes THEN
     Distance
     Time
     Speed  := Distance / Time
     (Speed > 0.05) AND (Speed < 2.00)
     INVOLVING(Distance, Time)
  ENDIF
```

Here the fields *Distance* and *Time* will be attached to the check.

3.6.4 Edit checks

An edit check describes a relationship between a number of fields, auxfields, locals, or constants. It states what the logical situation should be. If the values of the fields involved do not satisfy the statement, an error is invoked.

Hard errors use the key word CHECK. Soft errors use the key word SIGNAL. CHECK is the default.

What happens after an error depends on the application at hand. The most common interviewing mode will immediately produce an error message on the screen. The interviewer cannot continue to the next field until the problem has been resolved. In a CHECK, this is done by changing one or more values of the fields involved.

In a signal either the interviewer can change the value of a field or suppress the edit. There are four modes of operation in Blaise and the exact error reporting behaviour depends on the mode. In the most common data editing mode, errors are reported by means of icons displayed on the screen. The data editor can view error messages if necessary but often will not need to do so. (See Chapter 6 for more information on modes of behaviour in the DEP.)

One of the properties of Blaise is the fact that edits are very effectively inserted into the interview. When an edit is invoked, a dialog pops up and the interviewer is presented with an error message and a list of fields involved. The interviewer can place the cursor on the offending field, and immediately jump to the field to correct it. The interviewer then presses the End key, and is placed at the appropriate spot to continue the interview. In this way, a problem can be cleared up immediately, reducing the need for post-collection cleanup. In a large data model there can be literally hundreds or thousands of uniquely defined edit checks.

Suggestions for defining edit checks

Even though Blaise is good at the mechanics of error correction in an interview, this does not help if the interviewer and the respondent cannot understand the edit statement. Defining edit checks that are appropriate for the interview is an art. Here is what you can do to ensure that edits are appropriate:

- Resist efforts to write too many or too complicated edits.
- Determine which edits are appropriate for interviewing. Some edits require a great deal of subject matter knowledge and not all of the interviewers will have that knowledge.
- It is possible to make an edit soft in an interview but hard in interactive editing. This should rarely be done but sometimes is appropriate.
- Carefully choose which fields are available for correction for each edit check.
- Use field names or field descriptions that will be easily understood when presented in a list of choices to jump to in an edit message.
- Design a standard way of presenting edit message text to the interviewer. Do this in consultation with interviewers.

- Train the interviewer to immediately recognise whether an edit check is hard or soft. Additionally you can use cue words like [HARD] or [SOFT] in the edit message.
- Don't put too many signals (soft edits) in the interview. One reason to conduct a survey is to measure variability, not second-guess it.
- Aim edit checks at specific data collection problems. For example, respondents may accidentally respond in euros instead of thousands of euros.

A point to keep in mind is that the checks have to be consistent. If you define conflicting checks, the system will keep reporting hard errors whatever the values are, and it will be impossible for the interviewer to continue.

Consider putting the following information in an edit:

- An edit number
- A succinct statement
- Computed values

Consider assigning every check and every signal a unique number. This serves as an edit identifier. If there are problems with the edit, the user can provide the edit number and you can easily locate the edit in the source code.

When writing the error message, you should write a concise description of the problem, followed by a specific text for the interviewer to read. Also, use text enhancements and line feeds to make the message readable.

And lastly, use fills to display calculated values for clarification. For example, to display a wage rate (dollars per hour, when Salary and HoursWorked are the fields that have been collected.

Edit example

The following is an example of an edit statement:

```
SIGNAL
  (IHEAge <= 69) INVOLVING(DateOfBirth)
  "@E[WARNING EIHE16b-01] AGE OF HOUSEHOLD MEMBER, ^IHEAge, IS
     GREATER THAN 69.
     @/@/ I just want to confirm your age.  We have calculated
     your age as ^IHEAge. Is this correct?"
CHECK
```

First, *DateOfBirth* is asked and then *IHEAge* is calculated. Then a SIGNAL verifies whether the age of the respondent is less then 70.

WARNING indicated that the edit may be suppressed. *EIHE16b-01* is this edits identifier. The INVOLVING statement ensures that the *DateOfBirth* can be changed. *AGE OF HOUSEHOLD MEMBER* is the description of the problem, while the fill string *^IHEAge* displays the calculated value. The remaining text is for the interviewer to read back to the respondent.

Note that, by default, edits in Blaise are stated in terms of what is correct, not what is wrong. In other words, if the edit is true then by-pass the edit check or signal.

ERROR function

An alternative statement of an edit is with the ERROR function. You state what is wrong in an IF condition with the key word ERROR between the IF and ENDIF. For example:

```
IF (Distance/Time<= 0.05) OR (Distance/Time >= 20.0) THEN
    ERROR
ENDIF
```

The default edit statement convention is preferred by most Blaise users in keeping with the rest of the Blaise language, which states conditions in terms of what is supposed to happen. The default convention is more succinct and requires less code.

An edit check for numeric fields consists of arithmetical expressions that are compared by means of logical operators.

Arithmetical expression

An arithmetical expression is any expression that is composed of field names, auxfield names, local variable names, constants, and the arithmetical operators + (addition), — (subtraction), * (multiplication), / (division), DIV (integer division), MOD (remainder after integer division), and ** (power). Functions may be used to simplify computations and edit checks.

```
Income - Expenditures

MonthIncome * 12

Distance / TimeSpent

SQRT(Length ** 2 + Width ** 2)   {SQRT is a function.}
```

The names represent fields, auxfields, or locals. Enumerated type fields and set fields cannot be used in an arithmetical expression, unless they are used in a function that produces a numerical result.

String expressions

String fields may only be used in string expressions. You may only use the + (concatenation operator) and string functions with string expressions. Date and time fields may only be used in the special functions for these types. See the Table of Functions in the *Reference Manual* for a list of string functions.

Simple edits

A simple edit check is obtained by comparing two arithmetical expressions using one of the logical operators < (less than), <= (less than or equal to), > (greater than), >= (greater than or equal to), <> (unequal), = (equal), and IN (contained in).

```
Age > 15

Gender <> DONTKNOW

(Income - Expenditures) >= 0

Activity IN [Working, School]

Walking IN Transport
```

The first check states that *Age* should have a value greater than 15. The second condition is only true if the value of *Gender* is not equal to DONTKNOW. The third condition requires that the value of *Income* not be smaller than that of *Expenditures*. The fourth condition is only relevant for enumerated type fields. It states that the value of activity must be equal to one of the values in the set at the right-hand side. The type of the last condition is only used for set fields. It requires the value on the left side to be among the set of values recorded for the field *Transport*.

You can build more complex conditions with the logical operators AND, OR, and NOT. You can combine these operators to any depth.

Here are a few examples:

```
(Age > 18) AND (Age < 65)

(Job = Yes) OR (Job = No AND Looking = Yes)

NOT (Walking IN Transport)

(Value / Volume > 90) AND (Value / Volume < 110)
```

Rules of precedence

Conditions are evaluated according to the following rules of precedence:

Figure 3-2 Precedence of operators

Precedence	Operators	Type
1 (highest)	**	Binary
2	+, -, NOT	Unary
3	*, /, DIV, MOD	Binary
4	+, -	Binary
5	<, <=, >, >=, =, <>, IN	Binary
6	AND	Binary
7 (lowest)	OR	Binary

Unary operators take one operand, such as -X, whereas binary operators take two operands, such as X-Y. Operators with a higher precedence are evaluated before operators with a lower precedence. Use parentheses to enforce a different evaluation order or to improve readability.

Placement of edit checks in data model

Edit checks can be included anywhere in the RULES section, but they are only executed after all fields and variables involved have been processed. It is good practice to include the check at a point after the last field or variable has received a value. This is to ensure that the timing of the edit check is correct.

The following is an example of a part of a data model with a check:

```
FIELDS
  Distance "How far is it to your work (in km)?" : 0..300
  Time     "How long does that take you (in minutes)?" : 0..200
RULES
  Distance
  Time
  Distance / Time < 2.00
```

The DEP will generate an error message if the edit is violated. Here, the text of the error message is a copy of the check instruction. Particularly complex checks will be very hard to understand by the user. To make the error message more readable and instructive, it is possible to attach texts to both IF conditions and edit statements. The program will display the relevant text as the error message. In the following example, the check the error message will be identical to the check specification:

```
Distance / Time < 2.00
```

But for the check in the next example, the error message will be 'You cannot travel that fast!'

```
Distance / Time < 2.00 "You cannot travel that fast!"
```

Variable error message

We already mentioned that you can include variable text fills in question text. This is also possible in edit text. Take a look at the following data model:

```
FIELDS
  HomeTown "Where do you live?" : STRING[20]
  WorkTown "Where do you work?" : STRING[20]
  Distance
   "How far is it to your work (in kilometres)?" : 0..300
  Time "How long does that take you (in minutes)? : 0..200

RULES
  HomeTown
  WorkTown
  Distance
  Time
  Distance / Time < 2.00  "You cannot travel that fast from ^HomeTown to
        ^WorkTown!"
```

Suppose the field *HomeTown* contains the value 'Arlington' and the field *WorkTown* contains the value 'Rockville.' If the ratio of *Distance* and *Time* violates the condition, the resulting error message is 'You cannot travel that fast from Arlington to Rockville!'

Criteria for use of checks

To be sure the error messages will be reported on the data entry screen, there are some important rules concerning the use of checks:

- There must always be at least one field or auxfield involved in the check.
- One of the fields or auxfields involved should have the method ASK.

The reason for the first rule is that the DEP always attaches error messages to fields or auxfields. If there are none involved in a check, an error message can never be displayed on the screen.

The second rule is required to get rid of errors. That is only possible if the user is able to land on and change the value of fields or auxfields.

INVOLVING function

You can attach the special function INVOLVING to an edit check. This function has a variable number of parameters. Each parameter must be the name of a field or auxfield. All fields you specify this way are considered to be part of the check, and therefore error messages are attached to these fields if an error is detected. The following example is an alternative version of the preceding check:

```
FIELDS
  Distance
    "How far is it to your work (in km)?" : 0..300
  Time
    "How long does that take you (minutes)?" : 0..200

LOCALS
  Speed: REAL

RULES
  Distance
  Time
  Speed  := Distance / Time
  Speed > 0.05 AND Speed < 2.00
    INVOLVING(Distance, Time)
    "You cannot travel at that speed! "
```

The fields *Distance* and *Time* are part of the check. In case of an error, the system will list them both as places to jump to.

Another reason to use the INVOLVING function is to explicitly order the fields in the dialog box that pops up when an edit is invoked. Fields are listed in the Data Entry Program error dialog in the reverse order they are mentioned in the INVOLVING statement.

Robust edit writing

Each edit check, whether a check or a signal, has a place in the Blaise data set. This is so Blaise can keep track of which edits are invoked and knows whether the form is clean or not. If you add or delete edits, this will cause data file incompatibility. In production this can be a problem. Even though you may think you have the survey thoroughly specified, it happens very often that you need to add or delete an edit. Chapter 7 explains how you can easily transfer data from an old definition to a new one, but if the number of edit checks is changed, the edit information will not be transferred. Among other ramifications, you may lose all of your suppressions for signals and forms that were formerly declared clean may now be considered suspect.

There are three ways to handle the addition or deletion of edits:

- Use the key word RESERVECHECK as described in the next bullet. This gives a way to maintain the data set definition despite adding or deleting edits.
- To add an edit to a production instrument if a RESERVECHECK is not available, use a procedure to state the edit. This can influence the cleanliness status of the form, but edit checks in procedures are not stored in the data set. Procedures are covered in Chapter 5.
- If you need to change the data set definition because edits have been added or deleted, you can use a simple Manipula set-up to update the data definition. You will, however, lose all suppressions. This is covered in Chapter 7.

Identifiable Edits

Blaise offers an easy way to gain access to the results of the edit checks and signals defined in the rules of a data model. To make this possible each edit must have an identifier. Blaise uses an enumerated field, defined in the FIELDS section, as the placeholder for the result of the edit.

To identify such a field Blaise uses the so-called type EDITTYPE. The type EDITTYPE is a predefined enumerated field with the following definition: Passed,

Hard, Soft, and Suppressed. You ‘link’ such an EDITTYPE field to an edit in the RULES section by using the pipe-symbol. For example:

```
DATAMODEL EditTypeExample
FIELDS
  Number1, Number2: INTEGER[2]
  EditResult: EDITTYPE
  WhySuppress: STRING[50]
RULES
  Number1
  Number2
  SIGNAL
    EditResult  | Number2>Number1 "Number2 should be greater
                  than Number1"
  IF EditResult=Suppressed THEN
    WhySuppress
  ENDIF
ENDMODEL
```

In the example above, the result of the edit ‘Number2>Number1’ will be stored in the field EditResult. If number2>number1 the value is Passed. If number2<=number1 the value is Soft unless the soft error is suppressed by the user. In the latter case, the value will become suppressed.

The term, so-called type, was used above because can define your own EDITTYPE. Defining your own EDITTYPE will overrule the predefined EDITTYPE of Blaise. This makes it possible to extend the type with new values. These new values can be used from within the error dialog to indicate why an error is suppressed. However, this feature is only available if your EDITTYPE has more than 4 values.

An example:

```
DATAMODEL EditTypeExample
TYPE
  EditType = (OK "Edit is Okay",
              HardError "Hard error",
              SoftError "Soft error",
              SuppressedDK (7) "Don't know which to correct",
              SuppressedRF  "Refused to solve edit",
              SuppressedOther "Other suppress reason")
FIELDS
  Number1, Number2: INTEGER[2]
  EditResult "Number2 should be greater than Number1": EditType
  WhySuppress: STRING[50]
RULES
  Number1
  Number2
  SIGNAL
    EditResult  | Number2>Number1
  IF EditResult=SuppressedOther THEN
     WhySuppress
  ENDIF
ENDMODEL
```

Blaise assumes that the *OK, HardError* and *SoftError* correspond to the first three values on the predefined EDITTYPE. In the example above you can indicate in the error dialog why the error is suppressed (*SuppressedDK, SuppressedRF* or *SuppressedOther*). Notice that the field text of the *EditResult* field in the example is used in the error text of the edit in addition to the text defined with the edit itself.

RESERVECHECK

Blaise offers a way to robustly add or delete edits through the RESERVECHECK. The purpose of this is to reserve an edit for later usage or to replace an obsolete edit. It can be considered a placeholder. Two or three RESERVECHECKs should be placed in the RULES section of the data model and in some blocks. How many you place in each block depends on how confident you are about the block. If it is a well-used and tested block, then you don't need a RESERVECHECK there. If it is a new, untested section of a questionnaire, then you should place at least a few RESERVECHECKs in it. For example:

```
RULES
  Edit1
  Edit2
  RESERVECHECK  {is a placeholder}
```

To add an edit in production, delete or comment out the RESERVECHECK and replace it with an edit:

```
RULES
  Edit1
  Edit2
  {RESERVECHECK}  {was a placeholder}
  Edit3           {new edit check}
```

To delete an edit, comment out the edit and replace it with a RESERVECHECK:

```
RULES
  {Edit1}        {obsolete edit}
  RESERVECHECK   {new placeholder for Edit1}
  Edit2
  {RESERVECHECK} {was a placeholder}
  Edit3          {new edit check}
```

Make sure that you do not change the order of the edits.

For an example of edit checks see `editrule.bla` in `\Doc\Chapter3` under the Blaise system folder.

Toggling edit severity

On some occasions there may be a situation where it is wise to make an edit soft in an interview and hard during post-collection data editing. To do this, you use an IF-THEN condition to toggle the hardness of the edit. Make sure you reset the toggle the way you want it after the construction. For example:

```
IF CADI THEN
  CHECK
ELSE
  SIGNAL
ENDIF

MinWage > 3.50

CHECK
```

Note that CHECK and SIGNAL may be included in the part between THEN and ENDIF. This is the case in the example above. The switch to soft checking will only occur when the *CAPI* reserved word is TRUE, but will not occur when the *CADI* reserved word is TRUE. The *Reference Manual* provides more detail on the use of the *CAPI* and *CADI* reserved words. To be sure the system always

continues in check mode after the minimum wage edit check, we included CHECK after the edit.

3.6.5 Computations

Computations may be necessary to determine the proper route for processing the fields, to carry out complex checks, or to derive the value of a field if the Data Entry Program does not ask for a value. You can carry out computations anywhere in the RULES section. The general scheme is as follows:

```
Name  := Expression
```

Note the ':=' in the example above. *Name = Expression* would be interpreted as an edit check that the field *Name* has the same value as the field *Expression*.

The expression must be of the same type as the field *Name*. A numeric field expects a numeric expression and a string field expects a string expression. If the expression yields a value outside the valid domain of the field, the assignment will not be carried out and the field or variable will keep its old value. Fields that are computed need not be mentioned in route instructions. Blaise will apply the method KEEP by default. Therefore, you may omit the text *Speed.*KEEP.

```
FIELDS
  Distance "How far is it to your work (in km)?" : 0..300
  Time     "How long does that take you (minutes)?" : 0..200
  Speed    "What is your average speed (km/hour)?" : 0.0..100.0

RULES
  Distance
  Time
  {Speed.KEEP}
  Speed  := 60 * Distance / Time
```

Fields as expressions

There is a subtle difference between the following two statements:

```
A  := B
A  := (B)
```

The first statement is a straight field computation. The second statement is a field computation of an expression. That is, the parentheses ( ) make the right side of the computation an expression. The impact of this is that statuses do not transfer

in the latter case. In the second expression, if there are statuses of DONTKNOW or REFUSAL in the field *B*, then these statuses will not transfer to *A*.

When fields, defined as a block, are use in an assignment, only the values of the block are assigned. Remarks that have been added, will not (as the default) be copied. A toggle in the mode library allows you to specify if you want to copy the remarks while assigning a field to another field in the rules of your data model. The option can be set in the *Check Behaviour* section on the Advanced Toggles tab of the mode library editor. This applies to simple fields and to block fields. (See *Chapter 6, Data Entry Program*, the section on the mode library file.)

Assignment of statuses

Statuses are attached to each field and are known to you as DONTKNOW, REFUSAL, and EMPTY. When you make a field assignment, statuses will transfer from one field to another. However, it is possible to get imputation errors. Consider:

```
{Situation 1}

FIELDS
  S1 : STRING[10], DK, RF
  S2 : STRING[20], DK, RF
RULES
  S1  := S2

{Situation 2}
FIELDS
  S1 : STRING[10]
  S2 : STRING[20] , DK, RF
RULES
  S1  := S2
```

In situation 1, all statuses will transfer from *S2* to *S1*. However, in situation 2, if *S2* has either the REFUSAL or DONTKNOW status, an imputation error will occur.

Refer to the *Reference Manual* for full details on expressions and the use of standard functions.

3.6.6 Looping through rules

Consider the following fields:

```
FIELDS
    HHSize "How many people live in this house?" : 0..10

    Person "What is your name?" : ARRAY [1..10] OF STRING[30], EMPTY
```

You need a special type of rule for handling arrays of fields. For that, Blaise offers you the FOR loop. The general form of the FOR loop is:

```
FOR ControlVar  := LowerBound TO UpperBound DO
  Rule1 Rule2 ... RuleN
ENDDO
```

There are four reserved words: FOR, TO, DO, and ENDDO. Use the name *ControlVar* for a local that you use as a control variable in the FOR loop. It is defined in the LOCALS section of the current block. Its type must be integer. *LowerBound* and *UpperBound* must be either an integer constant or the name of an integer field (or integer auxfield).

The minimum value in the definition of the field *LowerBound* may not be smaller than the lower bound of the range of the processed arrays. The maximum value in the definition of the field *UpperBound* may not be greater than the upper bound of the range of the arrays.

The actual values of *LowerBound* and *UpperBound* indicate the initial and the final value of the control variable. *Rule1 Rule2 ... RuleN* represent the rules that are carried out for every value of the control variable.

To loop through an array for a dynamically determined upper bound you can use the value of an integer field as the upper bound. The following shows an example of such a construction:

```
RULES
  HHSize
  FOR I := 1 TO HHSize DO
    Person[I]
  ENDDO
```

The field *HHSize* contains the number of members in the household. The number of times the rules between DO and ENDDO are executed is given by the value of

HHSize. For example, if *HHSize* has the value *2*, only *Person[1]*, and *Person[2]* are processed. *HHSize* must have an appropriately defined integer range. In this example that would be *1..10*.

You may include any rule instruction between DO and ENDDO. For example, you do not always know how many array elements are needed. You need a different way to leave the array when the last element is reached. This can be done with an IF condition within the FOR loop. For example:

```
FOR I  := 1 TO 10 DO
  IF (Person[I-1] <> EMPTY) OR (I = 1) THEN
    Person[I]
  ENDIF
ENDDO
```

Here, *Person[1]* is always asked. If *Person[1]* is not EMPTY then *Person[2]* is asked. To leave the array, the interviewer leaves one of the *Person[I]* fields EMPTY. This example assumes that the field *Person[I]* is allowed to be EMPTY.

The following example contains a data model with a mixture of these instructions. The data model records the total distance from home to work, and also the distance covered by each means of transport. It checks whether the sum of the individual distances is equal to the total distance. The sum of the individual distances is computed in the local variable *Total*. This variable is set to zero before the loop starts. You do not need to include the instruction *Total := 0*, because Blaise automatically initialises local variables to zero values or empty strings every time the checking mechanism is invoked. In each cycle through the loop the specific individual distance is added to the value of *Total*.

```
DATAMODEL Commute5 "The National Commuter Survey"

LOCALS
  I, Total : INTEGER

FIELDS
  Name        "What is your name?" : STRING[20]
  Town        "In which town do you live?" : STRING[20]
  Job         "Do you have a job?" : (Yes, No)
  TotalDist   "What is the total distance to your work?" : 0..300
  NTransport  "How many means of transportation do you use?" : 1..5
  Means       "What is means of transportation no. ^I?": ARRAY[1..5] OF
    (Bus    "Public bus, tram or metro",
    Train   "Train",
    Car     "Car or motorcycle",
    Bicycle "Bicycle",
    Walk    "Walk",
    Other   "Other means of transportation")

  Distance    "What distance do you cover with means of transportation
              no. ^I?" : ARRAY[1..5] OF 0..300
RULES
  Name
  Town
  Job
  IF Job = Yes THEN
    TotalDist
    NTransport

    FOR I :=1 TO NTransport DO
      Means[I]
      Distance[I]
      IF Means[I]=Walk THEN
        Distance[I]<10
      ENDIF
      Total  := Total + Distance[I]
    ENDDO
    IF Distance[NTransport] = RESPONSE THEN
     (TotalDist = Total)
      "The individual distances don't add up to the total ^TotalDist"
    ENDIF
  ENDIF {Job = yes}

ENDMODEL
```

Please pay attention to the way the CHECK has been formulated, as in the following:

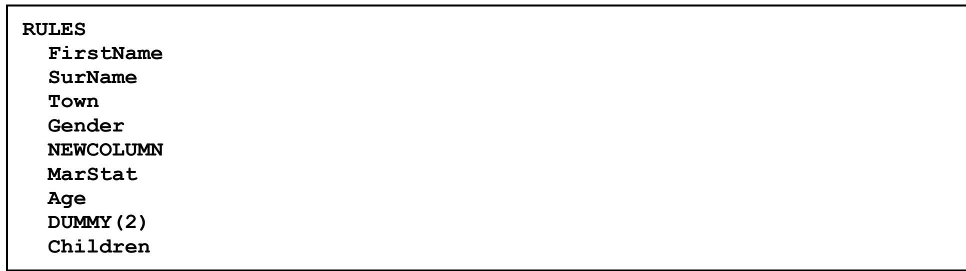
```
IF Distance[NTransport] = RESPONSE THEN
  (TotalDist = Total)
  "The individual distances don't add up to the total ^TotalDist"
ENDIF
```

Chapter 4 demonstrates how to array a block and how you can loop through the block array. Use of arrayed blocks is almost always preferable to looping through individual fields.

Without reference to the field *Distance[NTransport]*, the check would have been carried out straightaway after processing the field *TotalDist*. At that point, *Total* would still have the value zero, so that this always would have led to an error message. By including the reference to the field *Distance[NTransport]*, you instruct the system to carry out the check after the final field has been processed.

3.6.7 Layout elements in the RULES section

Four layout elements are available in the RULES section. They are DUMMY, NEWPAGE, NEWCOLUMN, and NEWLINE. Layout elements affect the placement of data entry cells in the FormPane (also known as the page). For example, DUMMY places a space between two adjoining cells in the page. A NEWPAGE preceding fields in the RULES section places succeeding cells on the next page. The NEWCOLUMN places succeeding cells in the next column if space is available. If space is not available, following cells are placed on the next page. NEWLINE is used only where horizontal data entry is allowed, such as in a table. It places succeeding entry cells on the next line of the horizontal display. The next example is extracted from `format.bla` under `\Doc\Chapter3`.



```
RULES
  FirstName
  SurName
  Town
  Gender
  NEWCOLUMN
  MarStat
  Age
  DUMMY(2)
  Children
```

The NEWCOLUMN places *MarStat* and succeeding fields in the second column of the page. The DUMMY(2) places two dummy fields between *Age* and *Children*. If DUMMY had been used, there would be only one dummy field between *Age* and *Children.*

Layout elements in the RULES section are overruled if a LAYOUT section exists in the same block as the RULES section. If a LAYOUT section exists, you can use DUMMY, NEWPAGE, NEWCOLUMN, and NEWLINE in the LAYOUT section instead of in the RULES section. See Chapter 5 for details.

3.6.8 Rules or no rules

It is possible to omit the RULES section completely from your specification. You are allowed to have a data model with only a FIELDS section. In this case, Blaise will assume that the fields must be processed in the order in which they are defined in the FIELDS section. If you use Manipula to manipulate your data file, you do not need a processing order, so it is very convenient not to have to specify rules.

If there is no RULES section, Blaise will apply the ASK method to all fields. The following sample shows a correct specification of a data model:

```
FIELDS
  Name     "Name of the respondent" : STRING[20]
  Town     "Town of residence" : STRING[20]
  Gender   "Gender of respondent" : (Male, Female)
  Age      "Age of respondent" : 0..120
  MarStat  "Marital status of respondent" :
           (NevMarr  "Never married",
            Married  "Married",
            Divorced "Divorced",
            Widowed  "Widowed")

ENDMODEL
```

Such a specification is useful if you want to document a file in which there are no specific instructions for the order of the fields. Blaise assumes the order to be the order in the FIELDS section.

3.6.9 Empty RULES section

It is also possible to have an empty RULES section. To do this, you still include the reserved word RULES, but you do not specify any rules. For such a model, all fields will be processed with the KEEP method. This is a consequence of the fact that you need not specify all fields in the RULES section. Blaise automatically KEEPs unmentioned fields. Thus, an empty RULES section will cause all fields to be processed with KEEP.

3.7 SETTINGS Section

You can specify some properties of your data model in a SETTINGS section. There are several types of settings. In this chapter we discuss two kinds of settings: key fields and languages. The ATTRIBUTES setting is discussed above in the section on *Answer Attributes*.

You must include the settings directly after the key word DATAMODEL and model identification before any other section. The following is an example of a SETTINGS section:

```
DATAMODEL Commute "The National Commuter Survey"

  PRIMARY
    Ident

  SECONDARY
    Manage.FormStatus

  ATTRIBUTES = REFUSAL, DONTKNOW

  LANGUAGES =
    ENG "English",
    NED "Nederlands"
```

In the above example, a field *Ident* is declared as a primary key. The field *FormStatus* in the management block *Manage* is declared as a secondary key, all fields in the data model have the attributes REFUSAL and DONTKNOW (unless otherwise specified at individual fields), and the instrument can handle both English and Dutch language texts. The *Ident* field could also be a block, in which case all fields of the block would be part of the key.

3.7.1 Key fields

Blaise allows two kinds of key fields. A primary key defines a unique identifier. A secondary key allows you to group forms together for more efficient processing.

Primary key

For most surveys, it is necessary to uniquely identify forms. This is done with a *primary key*. You can give any field the status of primary key. The primary key may be an elementary field, a block field, or a group of fields. You can always retrieve a form in the data set if you know its primary key value. To assign a field

the status of primary key, you specify the reserved word PRIMARY followed by the name of the field , as follows:

```
DATAMODEL Commute "The National Commuter Survey"

  PRIMARY
    Ident
FIELDS
  Ident "Identification number" : 10000..20000
  Name "What is your name?" : STRING[20]
  Birth "What is your date of birth?" : DATETYPE
RULES
  Ident
  Name
  Birth
ENDMODEL
```

Here the primary key field *Ident* is one elementary field. Entry of another form with the same number results in an error message. Consider the following:

```
DATAMODEL Commute "The National Commuter Survey"
  PRIMARY
    Region, Stratum
  FIELDS
    Region "Region of respondent." : 10..90
    Stratum "Stratum of respondent." : 1000..9000
```

Here the primary key is made up of two fields together.

Secondary keys

A *secondary key* is used to help retrieve and process groups of forms. You may filter such groups and give them special treatment in subsequent processing. This will speed up processing in many situations.

Two kinds of secondary keys are particularly valuable: to identify forms based on content (for example, all female respondents) and to identify forms based on administrative status (for example, complete versus incomplete interviews).

A secondary key can consist of a set of fields. You define such a key by specifying the reserved word SECONDARY followed by the list of fields together forming the secondary key. Commas must separate the items in this list. You can define more than one secondary key.

```
DATAMODEL Commute6 "The National Commuter Survey"

  PRIMARY
    Ident

  SECONDARY
    Complete
    GenderByJob = Gender, Job

  FIELDS
    Ident "Identification number." : 10000..20000
    Name "What is your name?" : STRING[30]
    Town "In which town do you live?" : STRING[20]
    Gender "What is your gender?" : (Male, Female)
    Job "Do you have a job?" : (Yes, No)
    Complete "Completion status." : (Done, NotDone)

  RULES
    Ident.KEEP
    Name
    Town
    Gender
    Job
    IF Job <> EMPTY THEN
      Complete  := Done
    ENDIF

ENDMODEL
```

The combination of *Gender* and *Job* is available as the secondary key *GenderByJob*. For example, you can select all cases of females with a job.

You can give secondary keys a name. Names are particularly convenient for combined keys.

```
SECONDARY
GenderByJob = Gender, Job
```

The secondary key *GenderByJob* is defined as a combination of *Gender* and *Job*. The Data Entry Program will present *GenderByJob* as one of the possible selection keys.

To use the trigram searching tool, add TRIGRAM to the definition of the secondary key. (The lookup method of trigrams is covered in Chapter 5.)

```
SECONDARY
  Name (TRIGRAM)
```

3.7.2 Languages

A data model can be multilingual. This is important for interviewing multilingual populations. To set up a data model to use different languages, you have to do two things. First, you specify the languages you want to use in the SETTINGS section. Second, you specify texts in these languages where necessary. For example, if you use English, French, and Dutch, you would include the following setting in the first part of your model:

```
LANGUAGES = ENG "English",
            FRA "Français",
            NED "Nederlands"
```

Each language specification consists of an identifier that is used by the system and a text that is included in the menu of the interviewing or data editing instrument.

Anywhere in your model where you can specify a text, you can now specify three texts in the same order as in the definition. For a question text, that could be:

```
Name "What is your name?"
     "Quel est votre nom?"
     "Wat is uw naam?"    : STRING[20]
```

If you specify fewer texts than the number of languages, the system will use the text of the first language for the unspecified texts. If you specify:

```
Name "What is your name?"
     "Quel est votre nom?" : STRING[20]
```

and you switch to Dutch, you will get the English text. It is possible to overrule the language order. You do that by putting the language identifier in front of the text:

```
Name "What is your name?"
     NED "Wat is uw naam?"   : STRING[20]
```

If you now switch to Dutch, you get Dutch, and if you switch to French, you get English.

To see a demonstration of a multilingual instrument, prepare the data model `commute4.bla` found in `\Doc\Chapter3` under the Blaise system folder. At this point you will have to switch languages through the menu system. A shortcut

key can be used to switch to the next language or to the previous language without going through the menus (see Chapter 6).

Question-by-question interviewer aids

You can use the concept of languages in different ways. One requirement for many surveys is the ability to have question-by-question interviewer aids attached to each question. These are known as Q-by-Qs. You can create a Help language that can be attached to a specific function key. This can be done as shown:

```
LANGUAGES = ENG "English",
            FRA "Français",
            HLP
```

Then in the FIELDS section:

```
Name "What is your name?"
     "Quel est votre nom?"
     "Any name will do.
      @/@/ N'importe quel nom est acceptable." : STRING [30]
```

Another way to implement question-by-question help is through the Windows® WinHelp utility. This is covered in Chapter 5.

Spoken and Unspoken language

Blaise has long had a language capability that allows interviewers to switch between spoken languages. The LANGUAGES declaration has also become a place for declaring unspoken languages with other uses, as shown in the following example:

```
LANGUAGES =
  ENG "English",          {spoken}
  FRA "French",           {spoken}
  HLP "Help"              {unspoken}
  MML "Multimedia",       {unspoken}
  MDL "Metadatalanguage"  {unspoken}
```

The example above illustrates that there can be spoken and unspoken languages. In the project *Datamodel properties*, you can designate which languages can be turned off for the interviewers.

In this example, two spoken and three unspoken languages are declared. Each has a 3-character identifier and a description between quotes. The identifiers such as ENG or HLP have no meaning to Blaise, except that these identifiers may be used

later in the source code. In the developer's environment, it is possible to state which languages are spoken and which are not. Spoken languages are available to interviewers, while they never know about the unspoken ones. A function key can be used to toggle between spoken languages during the interview.

The unspoken languages are used for other reasons, including multimedia questions (sound, images, video), Microsoft® WinHelp links for question-by-question help, or as a repository for additional field- or block-level metadata.

Blaise knows which language is in use. You can state IF conditions based on a language. For example, a text fill might be computed one way for English and another way for French.

3.7.3 TLANGUAGE, a provided language type

If multiple languages are defined in the settings paragraph of the data model, then Blaise provides a type called TLANGUAGE. This can be used to switch languages automatically between respondents in a multilingual household. For example, in a field of a household roster, you can ask what the preferred language is for each respondent. Then when arriving at the questions for the respondent with a different language, the language will automatically switch. Switching can be made dependent on the language abilities of the interviewer and whether a proxy can answer the questions. The name TLANGUAGE should not appear in a TYPE section or a type library. An example data model is `ncs05.bla` in `\Doc\Chapter5`.

If you have the following language definition:

```
DATAMODEL NCS05 "National Commuter Survey, example 5."

LANGUAGES =
  ENG "English",
  FRA "Français",
  NED "Nederlands"
```

A field can be defined with the type TLANGUAGE:

```
FIELDS
 . . .
  Language "What is your language preference?" : TLanguage
```

The respondent could choose the appropriate language. Later in the data model, the instruction SETLANGUAGE can be used to change languages. The

SETLANGUAGE instruction can be hard coded or it can be set by reference to an elementary field:

```
SETLANGUAGE(NED)
SETLANGUAGE(Language)
```

You can also reference an element in an array block. This is most valuable when dealing with multi-level rostering, as follows:

```
SETLANGUAGE(Respondent[n].Language)
```

In this situation Blaise will know which respondent to refer to according to the instructions you have programmed in the rules.

Switching languages in the interview

The interviewer can always switch languages by menu (or by hot key if provided for by the developer). Alternatively, the use of the type TLANGUAGE does the switching automatically in appropriate developer-defined situations. You should make sure that the automatic language switching described here and the language switching done through the menu do not conflict. For example, if the automatic language switching with SETLANGUAGE is done without condition and the interviewer switches languages through the menu, then the next time the checking mechanism is invoked the menu choice may be overridden because SETLANGUAGE was invoked again in the rules. You can avoid this confusion by embedding the SETLANGUAGE command in appropriate conditions.

Consistency between the type library and language setting

The languages used in the type library should be consistent with the language settings at the start of the data model. If your data model uses ENG and FRA, and your library uses ESP and NED, the data model will use Spanish for English and Dutch for French for the answer possibilities that come from the type library. The developer should ensure the necessary consistency.

Testing for the current language

You can test for the current language with the key word ACTIVELANGUAGE. For example:

```
IF ACTIVELANGUAGE = NED THEN
```

You might use the following example to determine which part of an external file to access or to properly construct some phrases, which are used as fills in language text:

```
  IF ACTIVELANGUAGE = ENG THEN
    Task  := 'wash your hands'
  ELSEIF ACTIVELANGUAGE = NED THEN
    Task  := 'uw handen gewassen'
  ENDIF
```

where *Task* may be a fill in a question text:

```
FIELDS
    HowOften "How often did you ^Task?"
             "Hoe vaak hebt u ^Task?" : 0..50
```

3.8 Functions

Blaise offers many functions that can greatly reduce the amount of programming you have to do. A function takes a field, auxfield, local, constant, or expression as an argument and returns an appropriate result. You can find a table of all available functions in the *Reference Manual*. A few representative function statements are given:

```
ABS(Difference)    {numeric function, absolute value}

ROUND(Total)       {numeric function, round to integer}

STR(Expenditure)   {numeric function, number to string}

LEN(AComputedString) {string function, length of string}

SYSTIME            {time function, system time}

SYSDATE            {date function, system date}
```

An example data model that illustrates date and time functions is `timedate.bla` found in `\Doc\Chapter3` under the Blaise system folder.

Handling of errors generated by functions

There are circumstances in which evaluating a function produces an error. In such cases Blaise assigns special predefined values to keep things running:

- Logarithms of 0 and of negative numbers are set to 0.
- Square roots of negative numbers are set to 0.
- Squares of very large numbers are set to the maximum possible number.
- Powers with very large exponents are set to the maximum possible number.
- Powers with very large negative exponents are set to 0.

Procedures and Dynamic Link Libraries (DLLs)

Blaise functions can handle most tasks. But you may occasionally need to program procedures within Blaise or external procedures in Dynamic Link Libraries (DLLs). These topics are covered in Chapter 5.

Functions and methods

In the Blaise documentation you will often see references to functions and methods. A method is applied to an entity, such as a field, with the dot notation. We have already discussed the ASK, SHOW, and KEEP methods that are applied to fields. Technically, methods are a kind of function, which is applied to an object through the dot notation.

There are a few places where you can choose either notation depending on the type of the argument. This is especially true of a few date and time functions.

3.9 Data File Compatibility

Blaise keeps a very close eye on whether the current data description is compatible with the current data set. If it is not and you try to invoke the DEP, you get a message stating that the data files are incompatible. This protects users from unintended consequences when changes are made to the metadata specification of the data model.

Even small changes in the metadata specification can cause data file incompatibility. You can handle incompatible data files in two ways. You can delete the data set and start over, but this is practical only early on in

development. You can also update the data set, which could be used while you are in development or during production.

3.9.1 Causes of data file incompatibility

The following is a list of ways the metadata definition can change. Many things make up the metadata definition and these must all be kept in mind:

- Change the number of fields.
- Change the order of fields in the FIELDS section.
- Change the valid range of even one field.
- Change the number of choices in an enumerated type.
- Change the number of edits (CHECKS or SIGNALS).
- Change the attributes of even one field.
- Change the primary key definition.
- Change the secondary key definition.

The following is a list of things you can change that will *not* affect the data definition:

- Change the name of a field.
- Change the name of an item in an enumerated type.
- Add or delete auxfields.
- Add or delete locals.
- Add or delete computations.
- Add or delete Blaise or alien procedures.
- Change an external file.
- Add, delete, or change text.
- Change an IF-THEN condition.
- Change a CHECK to a SIGNAL or vice versa.
- Modify a CHECK or SIGNAL as long as you do not add or delete one.
- Change a RESERVECHECK to a CHECK or a SIGNAL, or a CHECK or SIGNAL to RESERVECHECK.

If you are in production and you make changes to the instrument, you need to test the new instrument against the old data definition. Do this even if you do not think you have changed the metadata. The metadata definition can work very subtly at times. It is much easier to perform a one-minute test for compatibility than to hastily provide a fix on several hundred laptop computers when interviewers report problems. If the data definition has changed, you must provide a way for users to use the new instrument with the old data. This is covered in the following text and in Chapter 7.

Early development only

The first way of handling data set incompatibility is for developers in the Control Centre and is used only in the early development of an instrument. Choose *Database ➤ Data File Management ➤ Delete* from the menu (or Ctrl + D). A dialog box appears with a list of data files to erase. When you choose a data file, a prompt will be given to make sure you really mean to erase the file. If you accept, then the data files are erased. The next time you start the DEP, you will be prompted to create a new data set. Use this option only when developing. There is no elegant way to recover the data after erasing.

3.9.2 Production or development

When you are well along in the development process, data models can be very large and deleting data sets and starting over again to test the instruments can be time consuming. If you are in production, this method is absolutely inappropriate.

In these situations, you can handle data incompatibility by using Manipula. At this point we will cover a developer's strategy that makes use of a simple Manipula set-up. You do not have to understand anything about Manipula at this point, though you should not have any problem understanding the following set-up. The Manipula system is fully documented in Chapters 7 and 8.

Suppose you have an application `commute2.bla` in a working folder. You create a new folder under that called OLD. When you have created an instrument, have entered data, and are about to make a change to the metadata of the instrument, copy all instrument and data files to the directory `OLD`. This can be done easily by copying *.b* to `OLD`. Then invoke the Manipula set-up. It will move data from the old definition to the new. Since Blaise is based on a metadata approach, it will know field by field how to move data from the old data model to the new. It does this by matching field names. This is very powerful, especially when you insert questions in the middle of the instrument. Manipula will still be able to place data in the appropriate places in the new data model. This Manipula set-up would be:

```
USES
  CommuteOld 'Old\Commute2'
  CommuteNew 'Commute2'

INPUTFILE
  InFile : CommuteOld ('Old\Commute2', BLAISE)
  Outfile : CommuteNew ('Commute2', BLAISE)

MANIPULATE
  OutFile.WRITE
```

You create this file in the data editor of the Control Centre or through the Manipula Wizard. When you save the file, give it the extension `.man`. Perform a syntax check (prepare command) with F9 and run it with Ctrl-F9. This simple program will read data from the old to the new database no matter how complex the data models involved.

3.10 Good Programming Practices

The following is a list of the good programming practices mentioned in this chapter:

- Use indentation conventions when nesting IFs and ENDIFs.
- Make sure the IFs and ENDIFs line up. If some programmers do not follow this convention, then you will have a maintenance problem.
- Use comment braces {} to document your code throughout the specification.
- Use comment braces {} after ENDIFs that are widely separated from their IF statements so you know to which IF statement they belong.
- Place edit checks (checks and signals) among the routing of the fields in the RULES section. This will ensure that the checks will be invoked when you want them to be.
- Mix computations with the fields in the RULES section for the same reason you mix the checks and signals with the fields.
- Make sure a field or auxfield is part of the edit check statement. If this does not happen naturally, use the INVOLVING function.
- Re-declare a check after IF constructs where either a check or a signal may be invoked according to the situation found during the interview. This is so you are sure what severity the following edit checks will have:

- Give every check or signal a unique number. If there are problems in the field, the user can tell you which edit is wrong and you can easily find it to change it.
- Place RESERVECHECKs at frequent places in the RULES section to preserve robustness of data set definition.
- Type reserved words in capitals.
- Use very readable field names.
- For types in a type library or a TYPE section, use the convention that the type name starts with a capital *T* as in *TYesNo*. Thus you will always know this is a type.
- If a type is used in two or more places in one questionnaire, place the type in a TYPE section of that data model.
- If a type is used in two or more surveys, place the type in the type library.
- Always declare locals at the lowest level possible to make blocks independent and for performance reasons.
- Create and enforce a list of formatting standards for text displays in fields and edit checks (CHECKS and SIGNALS). These will include font, font size, colour coding, line spacing, and cue words for the interviewer and data editor.

3.11 Example Data Models

Following is a list of example data models and other files found in `\Doc\Chapter3` under the Blaise system folder. See the `read.me` file in this directory for any last minute changes. These data models illustrate the points made in this chapter. You can easily prepare and run them.

Figure 3-3: Example data models for Chapter 3

File Name	Description
commute1.bla	Basic data model, no blocks.
commute2.bla	Basic data model, two blocks.
commute3.bla	Basic hierarchical data model.
commute4.bla	Multilingual data model.
commute5.bla	FOR-DO loops.
commute6.bla	Primary and secondary keys.
timedate.bla	Time and date functions and methods.
types.bla	TYPE section and type library.
opentype.bla	An open-type field.
statuses.bla	REFUSAL, DONTKNOW, and EMPTY statuses and their converses.
format.bla	Text and FormPane formatting.
editrule.bla	CHECKS and SIGNALS.
showkeep.bla	Field METHODS.
ourtypes.lib	Type library (only 1 entry).
keepdemo.bla	Using KEEP to protect fields from review or update in confidential situations.
jumpback.bla	Enforced backward jumping.
filldemo.bla	Use of KEEP to enable a certain kind of variable text fill.
excroute.bla	Fields routed twice in exclusive branches of IF statement.
testcomp.bla	Data model where an imputation error is possible.
setcomp.bla	Showing computations, fills, and processing of SET fields.

4 Blocks and Tables

Blocks are a fundamental component of large or complex data models. They have many uses:

- Blocks add structure to the data model for clarity. It is easy to view blocks in the text editor. You can get overviews of the block structure at any level in the Structure Browser or Database Browser, or on paper via the metadata program Cameleon.
- Blocks form the basis for hierarchical (multi-level rostering) or relational instruments.
- Blocks are a type definition that can be reused. That is, blocks allow you to repeat groups of fields and their rules with just a few words. Thus a block can be used, and should often be used, as a kind of macro or subroutine.
- Blocks are the basis of selective rule processing. This allows you to have huge and complex data models that always run well while always enforcing all applicable rules.
- Blocks can be designated parallel, which allows you to break the normal interviewing sequence. With this feature you can implement concurrent interviewing, appointment and nonresponse blocks, or other kinds of blocks which should be available outside of the normal sequence of processing.
- Blocks are a logical unit for programming and testing. You can assign block names and tasks before programming and validate a list of blocks as they are completed.
- Blocks can be used as the basis for standardisation between surveys. You can include the same block of code in different instruments and use parameters to customise it.
- Programming code for blocks can be stored in a file separate from the main source file and brought into the data model file with the INCLUDE file statement. This allows many people to work on one data model at once without conflict.
- Blocks can be compiled into mini-data models for development and testing. If such blocks use parameters for reference to entities of other blocks, they can be plugged directly into larger operational data models. The use of these small data models has proved to be a powerful development and testing methodology.

- Blocks are the unit of data storage in the Blaise® data set. If no fields in the block are answered, then the block takes up no space in the data set.
- Blocks may form the basis of data readout. You can read out part of a data model for one client and another part for a different client with just a few words in a Manipula set-up.
- Blocks may form the basis of metadata management via the metadata program Cameleon. For example, you could give a SAS data set-up for one section of the questionnaire to one client and one for SPSS to another client.
- Blocks form the basis of a relational readout, which can be used by relational database systems.
- Blocks may form the basis of Manipula processing. This eliminates the need to read in the whole of every (large) form into memory when you need to read information from only a few blocks. This will speed up file processing.

Tables are a kind of block and are valuable because:

- Tables mimic paper forms for many household and economic surveys, especially for rostering.
- Tables increase data density in the FormPane, which is helpful to both interviewers and data editors. They can see more information in the Data Entry Program window.
- With large tables, you can scroll the FormPane like a spreadsheet.
- Navigation is facilitated, especially with the arrow keys and the page up and page down keys.
- Tables help organise long and complex instruments.
- Tables display both a block field name and an elementary field name in the FormPane (blocks alone just display the elementary field name). The block field name and the elementary field name can be replaced by field descriptions.

You can have extremely large and complex blocks and tables, but usually you will want to break them down into smaller units. In this section we discuss how to construct blocks and tables. Not all topics mentioned above are covered in this chapter; some are discussed in Chapter 5.

4.1 Blocks

You will see that blocks are an easy and natural way to program survey questionnaires. Many of the features mentioned above come automatically.

Block syntax

The syntax of a block is similar to that of a data model.

```
BLOCK BlockName
  TYPE                {optional}
  FIELDS              {optional, but usually present}
  LOCALS              {optional}
  AUXFIELDS           {optional}
  RULES               {optional, but usually present}
  LAYOUT              {optional}
ENDBLOCK

FIELDS {higher level}
  BlockFieldName : BlockName
```

Like a data model, it has a FIELDS section, a RULES section, possibly a TYPE section, a LOCALS section, a LAYOUT section, and so on. You can see that replacing BLOCK with DATA MODEL and ENDBLOCK with ENDMODEL would turn it into a data model.

A commonly used block is one that gathers information about a person as shown in the following example:

```
BLOCK BPerson "Demographic data of respondent"

  FIELDS
    Name     "What is your name?": STRING[20]
    Gender   "Are you male of female?": (Male, Female)
    MarStat  "What is your marital status?":
             (NevMarr "Never married",
              Married "Married",
              Divorced "Divorced",
              Widowed "Widowed")
    Children "How many children have you had?": 0..10
    Age      "What is your age?": 0..120

  RULES
    Name
    Gender
    MarStat
    IF Gender = Female THEN
      Children
    ENDIF
    Age
    IF (Age < 15)
     "Age of respondent is less than 15"
      THEN MarStat = NevMarr "he/she is too young to be married!"
    ENDIF
ENDBLOCK

FIELDS
  Person : BPerson
```

Terminology

In the example above, *BPerson* is known as a block type name and *Person* is known as a block field name.

Levels of organisation

The following example demonstrates the concept of levels of organisation that may be inherent in a data model with blocks:

```
DATAMODEL Example
  BLOCK BPerson
    FIELDS                          {block level}
      Name : STRING[20], EMPTY
      Age : 0..120
      Gender : (Male, Female)
    RULES                           {block level}
      Name
      IF Name <> EMPTY THEN
        Age
        Gender
      ENDIF
  ENDBLOCK

  FIELDS                            {datamodel level}
    Ident : 1000..9000
    HouseholdNumber : 0..25    Person : BPerson
  RULES                             {datamodel level}
    Ident
    HouseHoldNumber
    Person
ENDMODEL
```

We say that the fields *Ident* and *HouseholdNumber* are at the data model level and that the fields *Name*, *Age*, and *Gender* are at the block level. We also say that *Ident* and *HouseholdNumber* are at a higher level than *Name*, *Age*, and *Gender*.

Dot notation

To refer to a field or auxfield defined in a block from outside the block, you use the dot notation. For example, you can refer to the field *Name* from outside the *Person* block with the notation *Person.Name*.

A block allows you to bring together a group of fields that logically belong together. The block above contains fields that are necessary for establishing a person's identity and status.

Notice that we use the convention of the capital *B* as the first character of a block type name such as *BPerson*. By using this convention, it is always clear in later FIELDS sections when you are declaring a field in terms of a block type.

4.1.1 Blocks as types, repeating code

A block can be used as a type in a FIELDS section or a TYPE section. You can define fields in terms of a block name.

Processing such a block field in a RULES section means processing the block with all its fields and rules. The block as a type allows you to repeat much code with just a few words. In the following box, the block *BPerson* is repeated twice. This repetition is done with just a few words in a FIELDS section at a higher level.

```
BLOCK BPerson
  {Much programming code}
ENDBLOCK

FIELDS
  Person1 : BPerson
  Person2 : Bperson

RULES
  Person1
  Person2
```

All of the code for block *BPerson* is repeated twice, once for *Person1* and once for *Person2*.

Define an array of blocks

You can define an array of blocks. For example:

```
LOCALS
  I : INTEGER

BLOCK BPerson
  {Much programming code}
ENDBLOCK

FIELDS
  HouseholdNumber : 1..10
  Person : ARRAY [1..10] OF BPerson

RULES
  FOR I := 1 TO 10 DO
    IF HouseholdNumber <= I THEN
      Person[I]
    ENDIF
  ENDDO
```

Here, the block field *Person* is defined 10 times in terms of *BPerson* with the ARRAY statement. If there are six uniquely defined fields in the block, the block contributes 60 fields to the data model. When you repeat blocks you are repeating a lot of code as in a macro or a subroutine. In this code are data definitions in the FIELDS section and data relationships in the RULES section.

The code in the example below (which is also found in `\Doc\Chapter4\Commute7.bla`), demonstrates arrayed blocks that collect information for each person in a household. For each person with a job, an additional arrayed block collects information about the workplace. In this example the blocks are used without modification from person to person.

```
DATAMODEL Commute7 "National Commuter Survey, ex 7."

TYPE
  TYesNo = (Yes, No)

BLOCK BPerson "Demographic data of respondent"
  {Much code}
ENDBLOCK

BLOCK BWork "Data about work"
  {much code}
ENDBLOCK

LOCALS
  I: INTEGER

FIELDS                         {datamodel level}
  Street "Address of the household.": STRING[20]
  Town   "Address of the household, @/Town?": STRING[20]
  HHSize "Number of persons in the household?" : 1..10
  Person : ARRAY[1..10] OF BPerson
  Work :   ARRAY[1..10] of BWork;

RULES
  Street
  Town
  HHSize
  FOR I:= 1 TO HHSize DO
    Person[I]
    IF Person[I].Job = Yes THEN
      Work[I]
    ENDIF
  ENDDO
ENDMODEL
```

Enumerated instances of a block

There are times when it is appropriate to enumerate the repetitions of a block with carefully chosen names rather than use arrays. An example of code containing modes of transportation is in `\Doc\Chapter4\commute8.bla`, part of which is also shown in the following example:

```
DATAMODEL commute8 "National Commuter Survey, ex 8."
  BLOCK BDistance
    LOCALS
      AvgTime : REAL

    FIELDS
      Distance "Distance in this mode of travel?" : 0.0..200.0, DONTKNOW
      TotTime "What is the average time in this mode of travel?"
      : 1..200
      UnitTime "Unit of time?" :(Minutes, Hours)
      Minutes  "Answer in minutes" : 0..2000.0
    RULES
      {much code}
  ENDBLOCK

  FIELDS                       {higher level}
    TotDistance "Total distance traveled." : 0..999.9
    Car "@WCar or Car Pool.@W" : BDistance
    SubWay "@WSubway or light rail.@W" : BDistance
    Bus "@WBus.@W" : BDistance
    Walking "@WWalking" : BDistance
    Cycling "@WCycling." : BDistance
    Other "@WOther commuting mode." : BDistance
ENDMODEL
```

In this case, the interviewer would see names such as *Car.Distance* in the Data Entry Program if an edit were invoked. This name is easily interpreted and in this example is better than something like *Mode[1].Distance*.

Usually you want to apply repeated instances of the same block to different situations. For example, you may collect distance and time information for different modes of transportation. You would want to modify question text and apply different edit limits for each instance of the repeated block.

4.1.2 Block-level text

The easiest but most limited way to customise field text in a block is to add extra question text at the block level. In `\Doc\Chapter4\commute8.bla` above, each instance of the block is defined with some block-level text such as *'@WCar or Car Pool.@W'*:

```
Car "@WCar or Car Pool.@W" : BDistance
```

When the user is in the *Car* block, the text for the field *Distance* where *BDistance* is a block type name and the block field *Car* is an instance of the block type, will appear as shown in the following example:

```
Car or Car Pool
Distance in this mode of travel?
```

In the code for `commute8.bla` above, note the text enhancement of the block *Car* as opposed to the block *Walking*. In the former, the *@W* text enhancement applies only to the text at the block level since an ending *@W* matches the beginning one. In the block *Walking,* since there is no ending *@W,* the text enhancement will carry over to the field text also. Prepare the data model `commute8.bla` to see how this looks in practice.

In this example, text within the block is not modified as such but is preceded by the block-level text. For some applications this is sufficient and very easy to apply. For other applications there are more powerful methods.

4.1.3 Passing information to a block by direct reference

You can pass information to a block by directly referring to outside fields, auxfields, or locals from within the block. (You can also use parameters, which are covered below, or external files, which are covered in Chapter 5.)

From within the block you can refer to identifiers named in a direct higher level by using the identifier name. For example:

```
BLOCK BDistance
  FIELDS
    Distance "What is the distance you travel when you
             @B^ProperPhrase@B to work?" : 0.0..200.0
 ...

  RULES
    SIGNAL
    (AvgTime > Lower)
```

ProperPhrase and *Lower* may be names of fields, auxfields, or locals declared directly above the block. In this example they are defined at a direct higher level so you can use the elementary name of the entity without the dot notation.

Dot notation for field and auxfield names

If you need to refer to a field or auxfield from another block that is not a higher level block, you can use dot notation. For example, say you want to refer to the name of the person in your field text but that the name is gathered in a separate block. This can be accomplished as follows:

```
FIELDS
  Distance "^Person.Name, what is the distance you travel when you go to
  work?" : 0.0..200.0
```

Person.Name refers to the field *Name* in the block *Person*.

Choice between direct reference and parameters

Though direct reference to outside fields, auxfields, or locals from within the block is syntactically allowed, there are very good reasons for using parameters to accomplish the same task. See the section below on parameters.

Readable block names

It is very useful to choose readable names for block field names. This is primarily for the interviewer or data editor who must interpret these names in the Data Entry Program. They also make the code much more readable for the developer. In the examples above, either *Person1.Children* or *Person[1].Children* would be readable by the interviewer or data editor.

4.1.4 Two or more separate blocks

In the following, blocks *Person* and *Car* are considered separate blocks:

```
BLOCK BPerson
  FIELDS
    Distance "What is the distance to your main workplace?
              @/@/[NOTE] In kilometers!"  : 0.1..200.0
ENDBLOCK

FIELDS
  Person : BPerson

BLOCK BMode
  FIELDS
    Distance "What is the distance you travel when you
             Car or CarPool to your main work place?
             @/@/[NOTE] In kilometers!" : 0.0..200.0
ENDBLOCK

FIELDS
  Car : BMode

RULES
  Person
  Car
```

Edit checks between fields of different blocks

Edit checks between fields of different blocks may be written in the last defined block or at a higher level than the blocks. To maintain block independence (reusability) and performance, it is better to write edits between blocks outside of the blocks. Blaise® will invoke the edit in a timely manner.

If the block is repeated through an array statement, then you can use a FOR....DO loop to define the edit over all instances of the repeated block. Here you still have the advantage of writing the edit just once.

```
BLOCK BPerson
  FIELDS
    Distance
ENDBLOCK

FIELDS
  Person : BPerson

BLOCK BMode
  FIELDS
    Distance
ENDBLOCK

FIELDS
  Mode : ARRAY [1..6] OF BMode

RULES
  FOR I := 1 to 6 DO
    Mode[I].Distance < 1.5 * Person.Distance
  ENDDO
```

If you have a block that is repeated many times through enumeration and the edits are written outside the blocks, then there can be many similar edits, which have to be individually written. For example:

```
SIGNAL
  Car.Distance < 1.5 * Person.Distance
. . . .
   {Several other similar edits here.}
```

In the case of repeated blocks that are enumerated, it might be helpful to bring the edit check into the definition of the last block. This way, the edit is written just one time, which may ease maintenance. For example:

```
BLOCK BPerson
  FIELDS
    Distance
ENDBLOCK
FIELDS
  Person : BPerson

BLOCK BMode
  FIELDS
    Distance
  RULES
    SIGNAL
    Distance < 1.5 * Person.Distance

ENDBLOCK
FIELDS
  Car : BMode
  Bus : BMode
  Train  : BMode
  Other : BMode
```

The SIGNAL edit check is now within the last defined block. It directly refers to a field in another block through dot notation, namely *Person.Distance.* A better way to do this is to use an explicit parameter to bring the value of *Person.Distance* into the edit.

4.1.5 Nested blocks

It is possible to nest blocks. Just declare a new block within an existing block as shown in the following schematic:

```
DATAMODEL Nested
  FIELDS
    CommuteMethods
BLOCK BPerson
  FIELDS
    Name
    HaveJob
    CarPool
  BLOCK BJob
    FIELDS
      NameOfJob
      Distance
    RULES
      NameOfJob
      Distance
  ENDBLOCK {Bjob}
  FIELDS
    Job : BJob
  RULES
    Name
    HaveJob
    IF HaveJob = Yes THEN
      Job
    ENDIF
    IF Job.Distance > 10 THEN
      CarPool
    ENDIF
ENDBLOCK {BPerson}
FIELDS
  Person : BPerson
RULES
  Person
  IF Person.Job.Distance > 10 THEN
    CommuteMethods
  ENDIF
ENDMODEL
```

One way to develop hierarchical data models is through nested blocks.

Parent and child blocks

In the example above, the block *BPerson* is said to be a parent block of *BJob* and *BJob* is said to be a child of *BPerson*.

Dot notation for nested blocks

Within the block *BJob* you refer to the field *Distance* just by using its elementary name as in the extracted code in the following example:

```
RULES                           {RULES at BJob level}
  NameOfJob
  Distance
```

At one higher level, in the parent block *BPerson*, you refer to the same field with dot notation.

```
RULES                        {rules at BPerson level}
 . . .
IF Job.Distance > 10 THEN
  CarPool
ENDIF
```

At the data model level, to refer downward to the field *Distance*, use a dot notation with two dots and three names as in the following IF conditions:

```
RULES
  Person
  IF Person.Job.Distance > 10 THEN
    CommuteMethods
  ENDIF
```

Deeply nested blocks and edit display

You can nest blocks up to virtually unlimited levels. There is a way to limit the number of dotted block and field names that are displayed to the user (in either the *Active Signal* or *Hard Error* dialogs) when an edit is encountered in the Data Entry Program. You do this by editing the mode library file. This is covered in Chapter 6.

Separately coded subblocks

You can define a subblock outside a parent block but use it inside the parent block. Compare the following to the above:

```
BLOCK BJob
  FIELDS and RULES
ENDBLOCK {BJob}

BLOCK BPerson
  FIELDS
    Job : BJob
ENDBLOCK {BPerson}
```

The block *BJob* is defined outside of the block *BPerson* though it is used within *BPerson*. Whether you define a subblock within its surrounding or parent block is often a matter of style. However, if the block *BJob* is to be used in another block in this or another survey, then it is best to define it separately from the surrounding block. It is still possible for the nested block to have full knowledge

of information needed from the surrounding block. This is done with parameters, which are discussed in the following section.

4.2 Parameters

To pass information into or out of a block you can use parameters. Parameters are filled when the block is named in the RULES section. The explicit use of parameters is much more general than direct reference to outside information. They take a little more work to program but their use can pay big dividends. Advantages of parameters include:

- *Block independence*. You do not have to know ahead of time the names of fields or auxfields that are to be passed. This makes it easy to use the same block again in the same survey or in different surveys without modification.
- *Clarity of code*. You see at the head of the block which values are imported and which are exported.
- *Testing*. You can easily take even deeply nested blocks out of large instruments and give them to clients in small test instruments. The clients can experiment and quickly change the input values and see the effect on the block and the values of the outputs. If the block works in the small test instrument, it will work in the large one as well. See in `\Doc\Chapter4\test14.bla` for an example where a block nested within a table is brought into a small test instrument.

Because it is good programming practice to use parameters, further data models in this manual will use them.

4.2.1 Parameter example

An example with five import parameters in the block type *BDistance* is illustrated in the following example:

```
BLOCK BPerson
  FIELDS
    FirstName "What is your first name?" : STRING[20]
    SurName   "What is your surname?" : STRING[20]
  RULES
ENDBLOCK

FIELDS
  Person : BPerson

BLOCK BDistance
  PARAMETERS
    Respondent, ProperPhrase : STRING
    Lower, Upper : INTEGER

  LOCALS
    AvgTime : REAL

  FIELDS
    Distance "^Respondent, what is the distance you travel when you
    @B^ProperPhrase@B to your main work place?" : 0.0..200.0, DONTKNOW
    TotTime "What is the average time you spend on/in this mode of
    travel in minutes." : 1..200
    UnitTime "What is the unit of time?"  :(Minutes, Hours)
  RULES
    Distance
    IF (Distance > 0) OR (Distance = DONTKNOW) THEN
      TotTime
      UnitTime
    SIGNAL
    IF Distance > 0 THEN
      AvgTime := Distance/(TotTime/60)
      AvgTime > Lower
      INVOLVING(Distance, TotTime, UnitTime)
        "@R[WARNING E1] Your rate of speed seems to be to slow.
         It is ^AvgTime kilometers per hour. Is this correct?"
      AvgTime < Upper
      INVOLVING(Distance, TotTime, UnitTime)
        "@R[WARNING E2] Your rate of speed seems to be to high.
         It is ^AvgTime kilometers per hour. Is this correct?"
    ENDIF
ENDBLOCK

FIELDS
  Car : BDistance

RULES
  Person
  WholeName := Person.FirstName + ' ' + Person.SurName
  Car(WholeName, 'take the car or carpool', 15, 100)
```

The first two import parameters in the *BDistance* block are *Respondent* and *ProperPhrase.* These are string parameters that modify question text. The last two, *Lower*, and *Upper* are numbers used in edits. This block can be used again without modification in this survey or in others. See `\Doc\Chapter4\commut11.bla` for an example.

An instance of the block type *BDistance* is defined for cars.

```
FIELDS
  Car : BDistance
```

Then in the rules it is called with an appropriate parameter list.

```
  WholeName := Person.FirstName + ' ' + Person.SurName
  Car(WholeName, 'take the car or carpool', 15, 100)
```

It is in the rules that the parameters get their values. Parameters can take many forms, including a local *WholeName*, an expression *'take the car or carpool,'* or numbers *15, 100*. In another instance of the block, different parameter values are used:

```
FIELDS
  Bus : BDistance
RULES
  Bus(WholeName, 'take the bus', 10, 35)
```

The last two parameters have changed to modify the block for buses instead of for cars.

4.2.2 Parameter details

Parameters are introduced in the first part of the block definition in a section starting with the reserved word PARAMETERS.

Parameter definition

A parameter definition consists of a series of parameter names separated by commas and followed by a colon, and a type definition. A parameter name is an identifier, and therefore has to obey the rules for identifiers. The type can be any predefined field type except *open*, the identifier of a user-defined type, or a type BLOCK. In the FIELDS or RULES section, a parameter can be used in the same way as a field name or variable, except that it cannot be asked, shown, or kept.

When you use a parameterised block in a RULES section, you have to specify the actual values of the parameters between parentheses after the block field name. This is known as a parameter list. In the statement *Car(WholeName, . . .)*, the value of the parameter *Respondent* is replaced by the string *WholeName*.

Kinds of parameters

There are three kinds of parameters: *import*, *export*, and *transit.* The following table defines them:

Figure 4-1: Types of parameters

Parameter Type	Description
Import	Use an *import parameter* to bring a field value into a block. You may not change its value inside the block.
Export	Use an *export parameter* to compute a value in a block, and to move that value out of the block. The parameter need not have a value before the block is processed. A possible initial value is disregarded.
Transit	A *transit parameter* is a combination of an import and an export parameter. Before accessing the block, the parameter must have an initial value. Its value may be changed inside the block. After the block has been executed, the parameter will have its new value.

You give the status IMPORT, EXPORT, or TRANSIT to a parameter by preceding its definition with one of these three reserved words. The default status is IMPORT. The parameter *Name* is an import parameter in the example above.

The following example contains an import and an export parameter:

```
TYPE
  TYesNo = (Yes, No)

BLOCK BWork
  PARAMETERS
    IMPORT Respondent  : STRING
    EXPORT Commut: TYesNo
  FIELDS
    Job "Does ^Respondent have a job?": TYesNo
    Dist "What is the distance to ^Name's work?" : 0..200
  RULES
    Job
    IF Job = Yes THEN
      Dist
      IF Dist > 10 THEN
        Commut:= Yes
      ELSE
        Commut:= No
      ENDIF
    ELSE
      Commut:= No
    ENDIF
ENDBLOCK

FIELDS
  NameHH   "What is the name of the head of the household?": STRING[20]
  Commuter "Is the head of the household a commuter?": TYesNo
  Work: BWork

RULES
  NameHH
  Work(NameHH, Commuter)
```

Name is an import parameter. Its value is only used as a fill in a question text. *Commut* is an export parameter. Before the block is executed, this parameter has no value. It is computed in the block. When the block has been processed, the field *Commuter* will have the value *Yes* or *No*. In this example, *Commuter* is never displayed on the screen (though it is kept in the database).

You specify the values to be given to the parameters in the RULES section where the block is called. The order of the parameters in this parameter list when calling a block field must be the same as the order of the declared parameters in the block.

There are two reasons to use specific types of parameters such as import. One is clarity. When the block is processed it is clear what the parameters can be used for. The value of a transit parameter can be changed, an export parameter that has no initial value can acquire a value, and an import parameter will not change inside the block. The second reason is mainly technical. The system knows that an import parameter does not change, so inside the block there is no need to keep track of changes. This will improve processing efficiency.

A specific advantage of using import parameters is that you can use them to pass expressions to a block, while export and transit parameters must be variable entities: fields, auxfields, or locals.

Selective checking

When you pass information from one block to another (no matter how you do it in the code), parameters will be used by Blaise to do this. If you do not declare parameters explicitly then Blaise will generate them for you. These undeclared parameters are known as internal parameters. Parameters and the administration of parameters (explicit or internal) are the basis of the selective checking mechanism. This is what allows you to have a huge data model with tens of thousands of questions, ask all appropriate fields, always enforce all necessary routes and checks, and not slow down.

It is possible to abuse this system of parameter administration. You can cause the Data Entry Program to constantly check the value of thousands of parameters at one time. This would slow down a large data model considerably in interview mode. This won't happen if you follow the following few simple guidelines.

These will not be necessary for small data models but are valuable for large ones.

- Give structure to the data model. This may happen naturally for multi-level rostering household surveys but not for some economic surveys where many blocks are all on the highest level. In this case, it will be helpful if you enclose related blocks within higher level blocks. Blaise uses the block structure of a data model to determine which blocks need to be checked.
- Declare locals and auxfields at the lowest level possible. By doing this you accomplish two things: block independence, and you do not pass parameters unnecessarily from one block to a lower one. This will save on parameter administration and improve performance.
- Declare external files at the lowest level. The reasons are the same as for above. This is especially valuable when you have very large arrays and need to modify each instance of an arrayed block through an external file.
- Use explicitly declared parameters to pass information to a block.

The selective checking mechanism works by keeping track of which blocks have to be checked. It knows this by keeping track of values of explicit and internal parameters.

4.3 Included Files

Once blocks, or even FIELDS and RULES sections, have been developed and tested, they can be used in other data models as well. The program code for the block can be held in a separate file and included in the main data model file. This reduces development time and encourages co-ordination between surveys. You can be sure that job information is defined and asked in exactly the same way in all surveys involved.

Blaise allows a data model specification to be distributed over a number of text files. One file must be the *primary file.* In this file you can refer to other files using the INCLUDE command. It is the primary file that is prepared.

There are some advantages to using the INCLUDE command:

- You can save a block of source code in its own file and include it in a variety of different surveys. In some cases it is worthwhile to save even part of a block (for example, a RULES section applicable to different blocks) in a separate file.
- You can allocate the programming of a large survey to several different people, each developing their code in separate files.

4.3.1 Format of the INCLUDE command

The format of the INCLUDE command is:

```
INCLUDE "FileName"
```

The command starts with the reserved word INCLUDE, followed by the file name between quotes. When necessary, you can include a path. For example, a hard-coded path:

```
INCLUDE "c:\library\person.inc"
```

This refers to the file `person.inc` in the `LIBRARY` folder. When Blaise is preparing a model specification in a primary file and it encounters an INCLUDE command, it continues by preparing the included file. When it reaches the end of the file, it returns to the primary file and continues where it stopped.

You may also use relative path to identify the location of a file. This allows for the movement of source code around on different network environments. For example:

```
INCLUDE "..\source\person.inc"
```

You can have many INCLUDE commands in one file. You can also have INCLUDE commands in files that are included. There is practically no limit to nesting the commands this way. For example, `\Doc\Chapter4\commut13.bla`:

```
DATAMODEL Commute13 "National Commuter Survey, ex 13."

  TYPE
    TYesNo = (yes, no)
  LOCALS
    WholeName : STRING
  FIELDS
    WorkPlace "What is the name of your main workplace?" : STRING[20]
  INCLUDE "BPerson.inc"
  INCLUDE "TDistanc.inc"

  RULES
    Person
    IF Person.Job = yes THEN
      WorkPlace
      WorkPlace := CAP(WorkPlace)
      WholeName := Person.FirstName + ' ' +
                   Person.SurName
      Commute(WholeName)
    ENDIF
ENDMODEL
```

In this data model are two INCLUDE statements. The file `tdistanc.inc` contains yet another INCLUDE statement for the file `bdistanc.inc`.

```
TABLE TCommute

  PARAMETERS
    Respondent : STRING

  INCLUDE "BDistanc.inc"   {row block}
 . . .
ENDTABLE
```

You can put INCLUDE commands in the source code at any point where you could start a new section, such as a FIELDS section, a TYPE section, and a block or table. You cannot have an INCLUDE command in the middle of a RULES section.

4.3.2 FIELDS and RULES sections in included files

Though it is common to use included files for whole blocks, it may be advantageous to use included files for FIELDS and RULES sections.

For example, some data models are developed for use in international settings. While Blaise does have an explicit language capability, another approach is to separate the FIELDS and RULES sections. If you do this, you can change the language of the question text only, and leave the field names and the already-tested rules alone.

You might also want to separate fields and rules in separate files if you allow subject matter specialists or clients to concentrate on question text wording and presentation. Developers can then concentrate on the flow and data relationships in the RULES section.

4.3.3 File name extensions

It is helpful if included files have standard names for the file extensions. In this manual we use the standard extension of `.inc` for files that are included in the main data model. Other specific extensions may be adopted, for example, `.lib` for type libraries or `.prc` for procedures. This makes it much easier to back up and identify the many files that sometimes go into a large and complex data model. Suggested file name extensions are listed in Chapter 2.

4.4 Tables

Questionnaires are often laid out to present groups of questions as a table or a grid. A household roster of members is a typical example. Every row stands for a member of the household. The columns denote characteristics like age, gender, marital status, and so on. Economic surveys make extensive use of tables: rows can be used for different categories of exported goods and there can be columns for quantities and values. Examples of tables in `\Doc\Chapter4` include `commut13.bla` and `\Doc\Chapter4\commut14.bla`. The former is an example of a table with enumerated rows, the latter with arrayed rows. Data models `\Doc\Chapter4\ncs02.bla` and `\Doc\Chapter4\ncs03.bla` show the use of tables in a rostering situation. Data models `\Doc\Chapter4\hh1.bla` through `\Doc\Chapter4\hh6.bla` show several different ways to collect household rosters.

A table is a special kind of block. The sample below is an example of a table, from `\Doc\Chapter4\commut15.bla`:

```
DATAMODEL Commut15 "National Commuter Survey, ex 15"

  LIBRARIES MyLib

  INCLUDE "BPerson.inc"

  TABLE BHousehold "Demographic data of respondent"
    LOCALS
      I: INTEGER
    FIELDS
      HHSize "Size of the household?": 1..8
      Person: ARRAY[1..8] of BPerson
    RULES
      HHSize
      FOR I:= 1 TO HHSize DO
        Person[I]
      ENDDO
  ENDTABLE

  FIELDS
    HouseHold: BHousehold
  RULES
    Household
ENDMODEL
```

You define a table in the same way as a block. Just substitute TABLE and ENDTABLE for BLOCK and ENDBLOCK. Each field in the FIELDS section of the table denotes a row of the table. In the example above, the field *HHSize* is a row of one column. There are eight rows of the person block with several columns. The figure below shows how this would appear in the Data Entry Program (DEP).

Figure 4-2: Example table in the DEP

	HHSize	SurName	Job	NumberJobs	Distance
HHSize	2				
Person[1]					
Person[2]					
Person[3]					
Person[4]					
Person[5]					
Person[6]					
Person[7]					
Person[8]					

If you want more than one column in your table, you must use blocks within the table as rows. The highest level block in the table is the row block. If there are subblocks within the row block then they are part of the same row. The total

number of fields in the block defines the number of columns. If a block contains a conditional route where either field *A* or fields *B* and *C* are processed, there will be three columns, although fields *A* and *B* will never be processed at the same time for the same person. Every possible field must have its own place in the table.

Tables with unequal rows

You can have a table with unequal row sizes. In `commut15.bla` above, the first row of the table has one column; the other rows have five columns. If you prepare the data model you will note that the first column heading is entitled *HHSize*, not *FirstName*.

Scrolling in tables

You can have many columns in your table. If there are more columns than can be displayed on the screen, then the table will scroll to the right as it is being filled in (like a spreadsheet). If there are more rows than can be displayed, then the table will scroll down as the rows are filled in.

4.4.1 Extremely large tables

Blaise can handle extremely large tables, but a large table can have performance problems if not properly handled. This is because Blaise considers the whole table to be one page even though only a small part of it is on the screen. It will redraw the whole page every time you enter an answer. It can take a long time to redraw the whole page in extremely large tables. You can avoid this problem in two ways.

Break a table into smaller tables

The first is to break the table into several smaller tables, for example 5 tables of 20 rows each. Array the tables just as you would array block rows within a table. For example:

```
TABLE TGroup
  BLOCK BPerson
  ENDBLOCK
  FIELDS
    Person : ARRAY [1..20] OF BPerson
ENDTABLE
FIELDS
  Group : ARRAY [1..5] OF TGroup
```

In the above example, there are 100 rows spread over 5 tables.

Page breaks in a table

Breaking a table into several tables is acceptable if the rows of the tables do not have to communicate with each other. Another way to improve performance in a table is to break the table into several pages. This can be done either through a configuration file or through language statements in the rules of the table.

There is a setting you can edit in the Blaise mode library file, which is a file you can customise to control layout in the Data Entry Program, that automatically breaks tables into several pages. By default it allows eight rows on a page. Information on editing the mode library file, or *modelib* for short, is in Chapter 6.

If you want to take matters into your own hands and control the page breaks in the table through language statements, first disable the setting mentioned above in the modelib file. Then use the NEWPAGE key word. Suppose a table has 40 rows and many columns. You can break up the table into 2 pages with perhaps 20 rows each by using NEWPAGE in the LAYOUT section of the table. For example:

```
RULES
  FOR I := 1 TO 40 DO
    CommuteMode[I]
  ENDDO

LAYOUT
  AFTER CommuteMode[20]
    NEWPAGE
```

Another benefit of implementing page breaks between rows of extremely large tables is that this allows you to page through a table with the page down or page up keys. Without the pagination, the page keys would get you out of the table to the screen immediately following or preceding the table.

Dummy fields

Another useful feature is the DUMMY instruction. A dummy takes the place of a field, but does not do anything. You can use a dummy to create a 'hole' in your table, so fields line up correctly in a column.

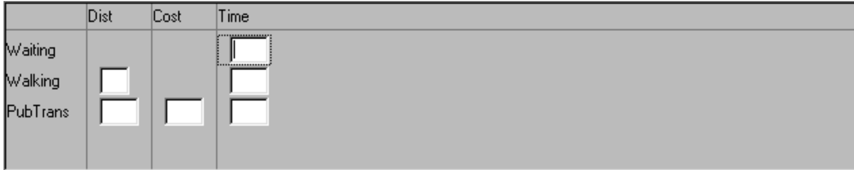
An example of the application of DUMMY is in the following example:

```
DATAMODEL Commut16 "National Commuter Survey, ex 16"

  BLOCK BWaiting
    FIELDS
      Time "How much time do you spend waiting?" : 0..500
    RULES
      Time
    LAYOUT
      AT BLOCKSTART
        DUMMY 2
  ENDBLOCK

  BLOCK BWalking
    FIELDS
      Dist "What distance do you cover walking?" : 0..20
      Time "How much time do you spend walking?" : 0..500
    RULES
      Dist Time
    LAYOUT
      AFTER Dist
        DUMMY
  ENDBLOCK

  BLOCK BPublicTransport
    FIELDS
      Dist "What distance do you cover by public transport?": 0..300
      Cost "What are the average daily costs?": 0..100
      Time "How much time do you spend in public transport?": 0..500
    RULES
      Dist  Cost Time
  ENDBLOCK

  TABLE BTransport "Transport data of respondent"
    FIELDS
      Waiting: BWaiting
      Walking: BWalking
      PubTrans: BPublicTransport
    RULES
      Waiting
      Walking
      PubTrans
  ENDTABLE

  FIELDS
    Transport: BTransport
  RULES
    Transport
ENDMODEL
```

The block type *BTransport* contains three block fields in the RULES section. Therefore, the table has three rows. Each row is a different block type and we want to know different information for the *BWaiting, BWalking,* and *BPublic Transport* blocks. But we also want our table to be well organised. Therefore, we have used the dummy field in the *Waiting* and *Walking* blocks. The result is that all these blocks have the same number of fields, either real or dummy. Thus the table has three even columns. The dummy fields are used in such a way that all

distances are put in column one, all costs in column two, and all times in column three. The following shows how this would look in the Data Entry Program:

Figure 4-3: Table with dummy fields in the DEP



4.4.2 Different kinds of tables

There are many different ways to construct a table. The examples below are for household enumeration and can be found in the `\Doc\Chapter4` folder.

Figure 4-4: Different kinds of household rosters

Method	Comment	Example File
The number of rows collected in the table depends on a previously collected household size.	To add more members to the household you must first return to and change the field collecting household size.	hh1.bla
Fill in rows of the household table until all members are included.	If you have previously collected household size, you can compare the number or rows in the table to the previous number.	hh2.bla
Collect just the names of all household members. Then collect details on all people.	This is done in one table in this example.	hh3.bla
Collect just the names of all household members. Then collect details on all people.	This is done in two tables in this example.	hh4.bla
Allow the interviewer to fill in the data of the household grid in any order.	Edits are used to ensure that the table is completely filled in.	hh5.bla

The illustrative examples above give only a flavour of what can be accomplished with various styles of tables. The one you choose depends on the requirements of the survey and the preferences of the study managers and clients. The code in all these examples is remarkably similar, as only a few changes are needed to alter the style of collection.

4.4.3 Protecting blocks and tables from further change

For some applications, you will want to protect the household section from further change once the data in it are collected and verified. This can be the case where a sampling routine is used to subsample household members for further questions. Changes to the household section after subsampling (and even perhaps after the subsampled respondents have answered the further questions) may cause the subsample to be redrawn, leading to inclusion of different members in the subsample. You might not want this to happen. In this situation, it is easy to disallow access back to the table after a certain point in the questionnaire. This is done with the KEEP or SHOW method applied to the table level. This method is illustrated in `\Doc\Chapter4\hh6.bla`. The basic syntax is shown in the following example:

```
FIELDS
  Done "Are you sure I have included everyone?" : (Yes, No)
TABLE BHHGrid
  BLOCK BPerson
    FIELDS
      Name : STRING[30], EMPTY
    RULES
  ENDBLOCK
  FIELDS
    Person : ARRAY [1..20] OF BPerson
  RULES
    FOR I := 1 TO 20 DO
      IF I = 1 OR Person[I - 1].Name <> EMPTY THEN
        Person[I]
      ENDIF
    ENDDO
ENDTABLE
FIELDS
  HHGrid : BHHGrid
RULES
  Done.KEEP
  IF Done = Yes THEN
    HHGrid.KEEP
  ELSE
    HHGrid
  ENDIF
  Done
```

In this example, the household table can be modified until the field *Done* receives the value *Yes*. After that, the interviewer can no longer get to the table. The data from the household table are retained and available to other parts of the program.

4.5 Mini-data Models

For large and complex surveys, it is extremely effective to use mini-data models for development and testing. The idea is to develop at least some blocks independently in a small data model, test them, and then plug them into a larger data model. The explicit use of parameters makes this possible. Once you know that the block works in the mini-data model, you know it will work correctly in the larger data model, assuming that the proper parameters are passed to it.

An example of a mini-data model is `\Doc\Chapter4\distance.bla`, which is a mini-data model for the block *BDistance.inc*.

4.6 Block Computations

There may be times when you have to transfer the contents of one block of data to another. If the blocks have the same block type definition, this can be easily accomplished with block computations. Consider:

```
BLOCK BDemoA
  FIELDS
    StrField: STRING[10], EMPTY, DK, RF
    IntField : INTEGER[2], EMPTY, DK, RF
    RealField : REAL[3], EMPTY, DK, RF
    EnumField : TYesNo, EMPTY, DK, RF
    SetField: SET OF TYesNoMaybe, EMPTY, DK, RF
ENDBLOCK
FIELDS
  Demo1 : BDemoA
  Demo2 : BDemoA
```

Block assignment

Demo1 and *Demo2* are two instances of the block type *BDemoA*. You can compute data from *Demo1* to *Demo2* with the following assignment in the RULES section:

```
Demo2 := Demo1                    {block computation}
```

It is almost always better to condition the block computation on a trigger field and enclose it within an IF condition. For example:

```
One_To_Two
IF ((One_To_Two = Yes) AND (Demo1 <> EMPTY)) THEN
  Demo2 := Demo1                  {Block computation.}
  One_To_Two := EMPTY
ENDIF
```

The block computation is done only when the value of the field *One_To_Two* is equal to *Yes*. This trigger field is set back to empty after the block computation to prevent inadvertent computations in case fields of *Demo1* are filled in again.

This example will copy values of all fields from *Demo1* to *Demo2* as well as all statuses such as *Don't Know*, *Refusal*, and *Empty*. This example is held in `\Doc\Chapter4\bcomp.bla`.

4.7 Array Methods

If you have an array of blocks or fields, you can insert, delete, or exchange array elements, even if the array elements are blocks. Consider the following (for the complete data model, see the example file `bcomp.bla`):

```
TABLE TBlockArray
  LOCALS
    I : INTEGER
  FIELDS
    Row : ARRAY [1..3] OF BDemoA
    InsertRow "Insert an empty row above this row number." : 1..3, EMPTY
    DeleteRow "Delete this row number." : 1..3, EMPTY
    ExchangeRows "Exchange these two row numbers."
                 : SET [2] OF (One, Two, Three), EMPTY
```

This table holds an array of three rows defined in terms of the block *BDemoA*. The fields *InsertRow*, *DeleteRow*, and *ExchangeRows* collect the number of the rows to insert, delete, or exchange. The actual manipulation of array elements is done in the RULES section:

```
  RULES
    FOR I := 1 TO 3 DO
      Row[I]
    ENDDO
    InsertRow
    IF InsertRow > 0 THEN
      Row.INSERT(InsertRow)               {Insert row.}
      InsertRow := EMPTY
    ENDIF
    DeleteRow
    IF DeleteRow > 0 THEN
      Row.DELETE(DeleteRow)               {Delete row}
      DeleteRow := EMPTY
    ENDIF
    ExchangeRows
    IF CARDINAL(ExchangeRows) = 2 THEN
      Row.EXCHANGE(ORD(ExchangeRows[1]),{Exchange rows}
                   ORD(ExchangeRows[2]))
      ExchangeRows := EMPTY
    ENDIF
    LandOn
ENDTABLE
```

For all three array methods, the trigger fields *InsertRow*, *DeleteRow*, and *ExchangeRows* are set to empty after the method has been applied. The array methods for insert, delete, and exchange are:

```
Row.INSERT(Index)
Row.DELETE(Index)
Row.EXCHANGE(Index1, Index2)
```

where *Index, Index1,* and *Index2* are numeric expressions. For insert and delete, *Index* is field *InsertRow* and *DeleteRow*, respectively. For exchange, *Index1* and *Index2* are elements of the set field *ExchangeRows*.

When insert is applied, all rows are moved down one line starting at the index number of the row. In this example of three rows, if *Index* is 2, then the second row becomes the third, and the previous third row disappears. The new second row is empty.

When delete is applied, the row of the index is emptied out, and all higher number rows move up one. If your intent is merely to empty out a row, but not to move other rows up one, where *n* is a number or numeric expression, use the following block computation:

```
Row[n] := EMPTY
```

For the exchange method, there are many ways to obtain the values of *Index1* and *Index2*. You can do this with integer fields, for example. In this example, a SET field is used to state them. This makes it easy to ensure that one number is not entered twice (Blaise checks this automatically for a SET field). The actual exchange will take place only when there are two numbers in the field *ExchangeRows*. This is ensured by the following statement in the IF condition:

```
IF CARDINAL(ExchangeRows) = 2 THEN
```

Then you obtain each entry from the set field *ExchangeRows* with the ORD function:

```
ORD(ExchangeRows[1]), ORD(ExchangeRows[2])
```

This gets the first and second values of the set field *ExchangeRows,* respectively.

4.8 Helpful Administrative and Survey Management Blocks

You can define blocks that have little or nothing to do with the subject matter of the survey but which help with the administration of the survey. For example, you might have blocks for:

- Nonresponse, to record reasons that the form is incomplete.
- Appointment, for appointments for one or all respondents.
- Unique identification, containing all components of the primary key.
- Screening, to make sure you only interview those who are supposed to be interviewed.
- Management, to keep track of interview progress, mode of processing, where data came from, who the interviewer was, and so forth.

- Data transmission, for Computer Assisted Personal Interviewing, so the interviewer can indicate to send forms once they are completed.
- Conclusion, to thank the respondent and to indicate when the subject matter part of the interview is done.

Not all surveys require all of the blocks above. By developing administrative blocks you can standardise procedures you use on various surveys and save much development time. An example is the National Commuter Survey, which is `\Doc\Chapter4\ncs01.bla`, and is shown in the following example:

```
DATAMODEL NCS01 "National Commuter Survey"

PRIMARY Ident
PARALLEL HouseNonResp
         Appointment_for_Household = HouseAppt
         HouseTransmit
LIBRARIES MyLib

INCLUDE "Manage.INC"     {Manage}
INCLUDE "DateTime.INC"   {DateTime}
INCLUDE "Ident.INC"      {Ident}
INCLUDE "Conclude.INC"   {HouseConclude}
INCLUDE "Transmit.INC"   {HouseTransmit}
INCLUDE "Appoint.INC"    {HouseAppoint}
INCLUDE "NonResp.INC"    {HouseNonResp}

  BLOCK BHousehold   {subs for real household block}
    FIELDS
      Done "Household roster finished?" : TContinue
    RULES
      Done
  ENDBLOCK
  FIELDS
    Household : BHousehold

  RULES
    Ident
    Manage
    DateTime
    Household
    HouseConclude('HOUSEHOLD')
    IF (Household.Done = EMPTY) AND
       (HouseConclude.ThankYou = EMPTY)  THEN
      HouseNonResp('HOUSEHOLD', HouseTransmit.DataReady)
      DUMMY
    ENDIF
    NEWPAGE
    IF (Household.Done = EMPTY) AND
       (HouseNonResp.Done = EMPTY) AND
       (HouseConclude.ThankYou = EMPTY)  THEN
      HouseAppt('HOUSEHOLD', Manage.FormStat)
      DUMMY
    ENDIF
    NEWPAGE
    HouseTransmit('', HouseConclude.Interviewer)
    IF Household.Done <> EMPTY  THEN
      HouseTransmit.DataReady := Ready
    ENDIF
ENDMODEL
```

This data model demonstrates how the administrative parts of a data model can work together. There are no subject matter questions in this data model yet. The only household-level question is a field called *Done* that indicates when the household roster part of the form is finished. This data model handles administration only for household-level data.

Two blocks, *DateTime* and *Manage,* will never be seen by the user. Other blocks are *parallel* blocks where you can break out of the normal sequence of interviewing. These include *Appointment_for_Household*, *HouseNonResp*, and *HouseTransmit*. Parallel blocks are covered below. The nonresponse and appointment blocks deserve special mention.

4.8.1 Nonresponse block

In case of nonresponse you will stop the interview without having answers to all relevant questions. For follow-up strategies in some surveys, it is necessary to collect information about the reason for nonresponse and the severity of refusals. You can put all questions that try to collect this information in a special block.

Here is the nonresponse block from the preceding `ncs01.bla` data model:

```
BLOCK BNonResp

  PARAMETERS
    IMPORT
      Whom : STRING
    EXPORT
      DataReady : DataReadiness
  LOCALS
    text1, text2 : STRING[199]
  FIELDS
    Thanks
      "Thank you for your time." : TContinue
    Reason
      " @Y[INTERVIEWER] Enter the reason for non-response for
        @B^Whom@B." : (Innac     "Impossible to contact",
         Refuses   "Refuses cooperation",
         NotPoss   "Cooperation not possible now, appointment not
                    possible within survey period",
         Others    "Other reason(s)")
    KindOfRefuse "Enter the severity of refusal."
      : (Mild, Firm "Firm but friendly", Hostile)
    OthReas
      " @Y[INTERVIEWER] Please enter the reason for non-response.@Y"
        : string[40]
    Done "Done with the nonresponse block. Enter 1." : TContinue
  RULES
    Reason
    IF Reason = Others THEN
      OthReas
    ENDIF
    IF Reason = Refuses THEN
      KindOfRefuse
    ENDIF
    IF Reason <> EMPTY THEN
      DataReady := Ready
    ENDIF
    Done
ENDBLOCK
```

The statements in the RULES section of the data model that call the nonresponse block are:

```
IF (Household.Done = EMPTY) AND
   (HouseConclude.ThankYou = EMPTY)  THEN
    HouseNonResp('HOUSEHOLD', HouseTransmit.DataReady)
ENDIF
```

Two parameters are used to customise the nonresponse block. Thus it must be put on the route in order to pass a value of the parameter. An additional advantage of having it on the route is to state when it should not be available. If the field *Household.Done* is filled, then the nonresponse block for the household itself should not be eligible. An export parameter is also defined here. It will compute a value of *Ready* in the field *DataReady* in the *HouseConclude* block.

4.8.2 Appointment block

Another reason to stop the interview is that people are willing to cooperate, but don't have time at that moment. You would like to have the option to make an appointment for another day and time. Questions for making an appointment can be included in a special block that gets the PARALLEL status. The implementation is similar to that of a nonresponse block, as demonstrated in `appoint.inc` in `\Doc\Chapter4`.

If you inspect the appointment block in its entirety you will see that the logic in the administrative blocks can be quite complex. This kind of programming requires extensive testing to ensure that the block will perform correctly in all situations. This goes not only for the inside of the block but also for its effect on other administration blocks. In this example, the appointment block is available only if the nonresponse block is empty.

All blocks are constructed using parameters. This allows the same block to be used as many times as necessary. For example, one instance of the appointment block may be used to make an appointment for the household and other instances of it may be used to make appointments for each of the respondents. In fact, four of the blocks for conclusion, transmission, appointment, and nonresponse will have multiple instances eventually. They will be used for individual respondents as well as household-level administration.

The blocks work together. For example, once the field *HouseConclude.Done* is filled in, the *HouseTransmit* block knows this through a computation.

The *Appointment_for_Household* block is available until the household roster is finished, or until the nonresponse block is filled in. To see this, prepare and invoke the `ncs01.bla` instrument, and then immediately access the *Parallel Blocks* dialog box using the menu command *Navigate ➤ Subforms*. You will see that the block *Appointment_for_Household* is available. Now return to the `ncs01.bla` main instrument. Fill in the field *Done* either in the household block or the nonresponse block. Now return to the *Parallel Blocks* dialog box. You will see that the block *Appointment_for_Household* is no longer available.

This `ncs01.bla` data model will be built up in this and successive chapters into a full demonstration of how to build a multi-level rostering, or hierarchical, household survey instrument, including all the survey management and data management tools.

4.9 Parallel Blocks

You often need to break out of the sequential processing order as specified in the RULES section. This is done with parallel blocks. Parallel blocks can be used for many situations, such as:

- *Concurrent interviewing*. One of the problems in interviewing respondents in large household surveys is that circumstances can change during the interview. Respondents may leave or join the session. With concurrent interviewing you can interview several respondents at once, interview respondents in any order, or proceed with one or a few respondents and come back to the others later.
- *Appointment block*. You never know when you will need to stop an interview and make an appointment.
- *Nonresponse block*. Because the respondent may decide at any time to quit the interview, you must be able to get to the nonresponse block at any time.
- *Form notes from previous surveys*. In panel surveys, form notes from previous waves may be valuable for subsequent interviewers. These can be made available for on-line inspection at any time during the interview.
- *Different respondents for different parts of one form*. In economic surveys, you may have to visit different respondents for each part of the form. With parallel blocks you can fill in whatever part of the form you need to, depending on whom you are talking to at the moment.

Parallel status

You can give the status PARALLEL to any block field. These are accessed in the Data Entry Program (DEP) by selecting *Navigate ➤ Subforms* from the menu, by selecting the appropriate parallel tab sheet, or through an assigned function key. This allows you to jump to a parallel block field at any time during a data entry session. If you reach the end of a parallel block, you can either finish the current form or jump back to the main instrument. You can jump from one parallel block to another. If you jump back to a parallel block that was interrupted, you can continue at the point of interruption.

Parallel blocks are introduced in the SETTINGS section in the beginning of the data model. You include the reserved word PARALLEL and a list of block field names. Here is an example:

```
PARALLEL
  HouseNonResp
  Appointment_for_Household = HouseAppt
  HouseTransmit
```

These are all names of administrative blocks. The second parallel block has been given a label, which will make the parallel block name more readable in the DEP.

Below is an example with nested blocks. If you want to jump to a nested block, you have to precede its name with the names of all surrounding blocks. The entries *Household.Head* and *Household.Partner* will be listed in the *Parallel Blocks* dialog box of the DEP. For deeply nested blocks particularly, such a long field name may be difficult to interpret. Therefore, it is a good idea to attach a name to the parallel name:

```
PARALLEL
  Head_of_household = Household.Head
  Partner_of_head   = Household.Partner
```

Now the entries *Head_of_household* and *Partner_of_head* will be included in the *Parallel Blocks* dialog box in the DEP.

The field name can also refer to an array of blocks. In that case, the individual array elements will also appear as parallel blocks in the DEP. Note that the *Parallel Blocks* dialog box will at any given moment contain only those field names that are accessible in the RULES section given the current state of affairs. You can experiment with the data model `ncs01.bla` above to see how the administrative blocks interact with one another.

Unrouted parallel blocks

Parallel blocks that are not mentioned in the rules are given the KEEP status. This means that you will never see them in the normal interviewing sequence. However, they are always available from the *Navigate* menu. An unrouted parallel block is the last block put on the route in the instrument. Other blocks will not have access to it since the fields in the unrouted parallel block will come after everything else. Thus if you wish to use a value from a parallel block in another part of the instrument, you will have to route the parallel block before the other blocks which will use it. You would do so with carefully considered IF conditions, or perhaps put a KEEP status on it.

Parameters and parallel blocks

Parallel blocks normally do not have to be put on the route in order to be accessible during the interview. However, if you pass parameters to a parallel block, then it is necessary to put it on the route. In these situations, you must carefully consider the routing conditions to keep the parallel block from coming on the route when you do not want it to do so. See the nonresponse and appointment examples below.

The code below shows that the nonresponse parallel block is on the route only when it makes sense to have it there.

```
IF (Household.Done = EMPTY) AND
   (HouseConclude.ThankYou = EMPTY) THEN
    HouseNonResp('HOUSEHOLD', HouseTransmit.DataReady)
ENDIF
```

Specifying text for parallel blocks in the DEP

You can specify text that will display in the *Parallel Blocks* dialog of the DEP. By default, when parallel blocks are selected in the DEP, the parallel block name or the identifier that is declared in the data model appears in the dialog box. You can specify more descriptive text for the parallel block on the *Data model Properties* dialog box, which is described in Chapter 3*,* section 3.8.

The names of parallels are stored in a file with a `.bxi` extension. In this file, the system keeps track of the specified text for the parallels. Each time you prepare the data model, the system adds or removes parallels, and keeps track of the text that was specified.

4.9.1 Blocks chosen by menu

Though the method of block access through parallel blocks is complete and easy to implement, there may be times when a sponsor will specify an alternative way to do the same thing. A way to provide access to particular blocks without using parallel blocks is to provide a menu at a field. At the menu field, the interviewer chooses the block to execute at that moment. The execution of the blocks is conditioned on the choice the interviewer makes in the menu field. A sample of the code for this follows, from `\Doc\Chapter4\menujump.bla`:

```
DATAMODEL MenuJump
  FIELDS
    Menu "Choose a module to execute now."
           : (Accountant "Accountant's module",
              Controller "Controller's module",
              Supervisor "Supervisor's module",
              Finished "Finished with enumeration")
  BLOCK BAccountant
      …
  ENDBLOCK
  BLOCK Bcontroller
      …
  ENDBLOCK
  BLOCK Bsupervisor
      …
  ENDBLOCK
  FIELDS
    Accountant : BAccountant
    Controller : BController
    Supervisor : BSupervisor
  RULES
    Menu
    IF Menu = Accountant THEN
      Accountant
      IF Accountant.Done = Continue THEN
        Menu := EMPTY
        Accountant.Done := EMPTY
      ENDIF
    ELSEIF Menu = Controller THEN
      Controller
      IF Controller.Done = Continue THEN
        Menu := EMPTY
        Controller.Done := EMPTY
      ENDIF
    ELSEIF Menu = Supervisor THEN
      Supervisor
      IF Supervisor.Done = Continue THEN
        Menu := EMPTY
        Supervisor.Done := EMPTY
      ENDIF
    ENDIF
    Accountant.KEEP
    Controller.KEEP
    Supervisor.KEEP
ENDMODEL
```

The IF-ELSEIF construction puts only one block on the route at a time. The interviewer completes the module, then moves back to the field *Menu* to select a new module. The KEEP statements after the IF-ELSEIF construction are necessary to avoid losing data as Blaise normally clears all off-route data from the database when the form is closed.

4.10 Hierarchical Data Models

Many survey instruments have a hierarchical structure. An obvious example is a household survey. You have the household at the highest level. Each household has a number of characteristics, like composition, income, address, and so on. Households consist of persons. Therefore, the second level in the hierarchy is the person, and each person has some characteristics, like age, gender, and marital status.

A hierarchical data model can have more than two levels. For example, the persons in the household survey have a number of jobs, and for each job, perhaps a number of locations. For each location, there might be several means of transport.

Blaise allows you to build hierarchical data models using several methods. These data models can have several or many levels.

- You can nest blocks within blocks.
- You can use arrays of blocks and arrays of blocks within other block arrays. You can associate elements of one array with elements of another array. You can do this in complex but very efficient ways if necessary.
- Parameters allow you to dynamically determine which values to pass into a particular arrayed block.
- Blaise keeps track of which fields are relevant to any situation. Thus, in an edit check, only the relevant fields are shown to the Data Entry Program user when the edits are encountered during a data entry session. The interviewer does not have to engage in complex navigation to fix a problem.

Hierarchical data model sample: The National Commuter Survey

The best way to learn how to build hierarchical data models is to inspect code. The data model `\Doc\Chapter4\ncs02.bla` is a realistic if incomplete example of a hierarchical household survey. This survey is called the National Commuter Survey. In this survey, names and personal data for up to 10 members of a household are gathered in a roster. Here we ask how many jobs outside the home each individual has. After the roster is completed, we collect information about the name of each employer for each respondent. For each respondent/employer category, we collect information about work locations. Depending on how you count, this is a three- or four-level rostering situation.

The household roster is collected in its entirety before detailed information about individuals and their work is collected. This means that the rostering in this example is not purely a matter of nested blocks.

Let's look at two separate but related arrays. The example here is for the household roster.

```
TABLE THousehold
  LOCALS
    I : INTEGER  {looping counter for arrayed table}
  FIELDS
    Person : ARRAY[1..10] OF BPerson
    Done "Household roster finished?" : TContinue

  RULES
    FOR I := 1 to Address.HHSize DO
      Person[I]
    ENDDO
    Done
ENDTABLE
FIELDS
  Household : THousehold
```

The row of the household roster table is contained in *BPerson*. This example allows up to 10 rows to be filled in, but only up to the value of *HHSize*.

The table here is for rosters of jobs and locations.

```
TABLE TPerson  {Information about the nth person's workplace.}
  PARAMETERS
    IMPORT
      Whom : STRING
      WorkNum : INTEGER

  LOCALS
    J : integer

  FIELDS
    Work : ARRAY[1..5] OF BWork
    Done "Interviewer, is the workplace roster finished?" : TContinue

  RULES
    FOR J := 1 to 5 DO
      IF J <= WorkNum THEN
        Work[J](Whom, J)
      ENDIF
    ENDDO
    Done

ENDTABLE
FIELDS
  PersonNum : ARRAY[1..10] OF TPerson
```

We see that the table *PersonNum* is arrayed up to 10 times. Within the table is an array of row blocks, and if you were to inspect the code even further (`bwork.inc` in `\Doc\Chapter4`), you would see that there is another block arrayed within the row block. This is how you can have arrays nested within arrays.

4.10.1 Connecting arrayed blocks

The arrayed tables *PersonNum[]* are not nested within the household roster table *Household*. Yet each table *PersonNum[]* must be associated with a corresponding row of *Household*. This is done in this case at the data model level with the following statements:

```
AUXFIELDS
  WorkNum : 0..10
  WorkName : STRING[30]
…

Household
FOR G := 1 TO 10 DO
  IF Household.Person[G].Job = YES THEN
    WorkNum := Household.Person[G].NumberJobs
    WholeName := Household.Person[G].FirstName + ' ' +
                 Household.Person[G].SurName
    PersonNum[G](WholeName, WorkNum)
  ENDIF
ENDDO
```

Each *PersonNum[G]* table is associated with a person in the household roster, namely, *Household.Person[G]*. We wish to pass information from the household roster to the work rosters and have chosen to do this with parameters. For convenience only, the Auxfields *WorkNum* and *WholeName* were created and computed before each *PersonNum[G]* was called.

The data model `\Doc\Chapter4\ncs03.bla` carries the hierarchical example even further. There are multiple instances of administrative blocks to take care of individuals as well as the household level. For example, you can have an appointment at the household level or at an individual level. It is possible to schedule more than one appointment at a time for individuals. The individual rosters are parallel blocks, which allow you to interview one person or several people at one time.

4.11 Selective Checking Mechanism and Instrument Performance

Blaise is designed to handle the world's largest and most complex surveys and has many features that allow it to do this. One of the main considerations when designing such a system is the proper handling of the administration of the rules, including routing, edits, IF conditions, and computations.

In a large instrument, there may be thousands of questions and thousands more rules that govern their implementation. In such a data model, the constant checking of all rules could overwhelm the computer. Blaise always checks all appropriate rules. It does not always check all rules, but just the appropriate ones.

In order to know which rules to enforce, Blaise employs a selective checking mechanism based on parameters and blocks. If the data model is set up properly, this selective checking mechanism works extremely well.

A very large instrument with thousands and tens of thousands of fields, edits, and computations can perform very well if it is programmed correctly. But if the same instrument were constructed naively, it probably will perform poorly.

The discussion below briefly describes the selective checking mechanism and then lists simple rules that, if followed, should result in an instrument that performs well.

4.11.1 Performance and parameters

Blaise uses parameters to optimise the performance of the checking mechanism during instrument use. By keeping track of parameters, both explicit and internal, Blaise knows which blocks to check.

Parameters are sometimes declared explicitly by the developer. Other times Blaise generates them. This happens when reference is made from within a block to outside the block without explicitly declaring parameters to do this. Parameters generated by Blaise are called *internal parameters.* You can see both internal and explicit parameters in the Structure Browser.

From a performance point of view, all parameters keep track of information that is being passed from one block to another. For example, in complex economic surveys, it is not unusual to have 10, 20, 30, or more parameters being administered for each block. If there are a lot of blocks, then it is possible that too many parameters are administered.

If you find that you have a large number of parameters being administered all at once, there are some things you can do that are extremely effective:

- Add structure to the instrument. Blaise uses structure to know which parameters to keep track of. That is, Blaise handles structured instruments optimally. This means nesting blocks. If you have many blocks on a high level, each with a lot of parameters, you can place some of them within other higher level blocks. The fewer high-level blocks the better.
- Declare all locals and auxfields at the lowest level possible. By doing this you accomplish two things: block independence, and you do not pass parameters unnecessarily from one block to a lower one. This will save on parameter administration and improve performance.
- Declare external fields from an external file at the lowest level possible. If done this way, there will be no parameter administration on external fields, which isn't needed because they are unchangeable.
- Sometimes you bring information into a block for the sole purpose of providing context to the interview. For example, you might want to display a respondent's name or identification number. You can use the field text at the block level to hold this information. This information will be displayed at the top of the InfoPane and will not be counted as a parameter that has to be administered. For example:

```
BLOCK  BTextDemo
  FIELDS
    TextTry "@/This is the question text.": STRING [1], EMPTY

ENDBLOCK
  FIELDS
    TextDemo "This is block level text for context.":
      BTextDemo
```

The question text display for the field *TextTry* will be:

```
This is block level text for context.

This is the question text.
```

- Use import parameters to pass information into a block if that information is not going to be changed. Blaise knows that import parameters do not need the same amount of administration as export or transit parameters.

- Be careful about accessing elements of an array. Explicitly declare array parameters and compute their value before calling the array. If array elements are not explicitly declared, Blaise will declare internal parameters for all possible array elements. This can mean that hundreds or thousands of unneeded parameters are generated and administered. By explicitly declaring parameters and computing their value before passing the values in this situation, you will greatly cut down on the number of parameters that Blaise keeps track of.
- Stop forward parameter checking with IF statements in the RULES section, especially for major blocks that follow the present block or field. Blaise always forward checks parameters that are currently on the route. If thousands of questions are on the route, then their parameters will be checked, even if the interview has not yet arrived at that point.
- If a field is used often in IF conditions, recast it once as a local, then use the local in the IF conditions. This is because Blaise has much easier access to the values of locals than to the values of fields.
- If there are edits between fields of different blocks, then put the edits at a higher level than the two blocks. The edit will be invoked in a timely manner and neither block will be burdened with parameter administration for the fields in the edit.

To view internal parameters, open the data model in the Structure Browser as described in Chapter 2. In the Structure Browser options, check to view internal parameters. They will then be marked in the tree of the model with the letters *GP*. You may find that you are passing a lot more information between blocks than you thought.

4.11.2 Other performance gains

Other performance improvements may be achieved for reasons having nothing to do with parameter administration.

- In a large table, you can break up the table with NEWPAGE in a LAYOUT section. This will keep Blaise from redrawing the entire table, even the part that is not displayed. A table of 100 rows may be better displayed with a NEWPAGE every 20 rows. The default mode library file divides tables into eight rows per page. For more information on the mode library file, see Chapter 6.

- Turn off the Ditto feature in the Data Entry Program configuration file if you do not need it. (See Chapter 6.) With this feature on, two copies of the data record are stored in memory.

4.12 Good Programming Practices

The following is a list of good programming practices that were mentioned in this chapter:

- Use blocks often and well to speed instrument development.
- Use blocks to repeat code like a macro or subroutine.
- Use readable block names to make choices readable for the interviewer or data editor when an edit is invoked.
- Use parameters when you need to pass values into and out of blocks in order to make the blocks very general and reusable.
- Use import parameters as much as possible to cut down on parameter administration.
- For large instruments, use blocks to add structure to the code. This will help performance.
- Declare locals, auxfields, and external files at the lowest level possible. This helps make blocks independent and improves efficiency.
- For extremely large tables, you may have to break up the table with NEWPAGE key words in order to improve performance. You may also consider breaking up extremely large tables into two or more tables by editing the modelib file.
- When you use INCLUDE statements, give the included files a standard extension.
- Edit checks and computations between separate blocks should usually be written at a higher level than the blocks. An exception is when you have repeated a block several times through enumeration. In this latter case, it may simplify maintenance if the edit is written within the last defined block.
- For your organisation, define special blocks that help with survey administration. These include appointment and nonresponse blocks. These blocks should work closely together. It is often helpful to test the instrument with just the administrative blocks in place to make sure that this part works the way you wish it to. See `ncs01.bla` for an example.

4.13 Example Data Models

The following is a list of example data models and other files found in `\Doc\Chapter4` under the Blaise system folder. See the `read.me` file in this folder for any last minute changes. These data models illustrate the points made in this chapter. You can easily prepare and view them.

You will have to prepare the `mylib.lib` type library before preparing some of these data models. There are several files in the folder with the `.inc` extension. These are included files for some of the later data models.

Figure 4-5: Example data models used in Chapter

File Name	Description
commute7.bla	Illustration with two simple blocks.
commute8.bla	One defined block repeated several times through enumeration. Repeated blocks are modified minimally with block-level text.
commute9.bla	Adds to commute8 by introducing parameters for some edits in the block.
commut10.bla	Adds to commute9 with parameters for text enhancement.
commut11.bla	Has two separately defined blocks, with edits between the blocks defined outside of the blocks.
commu11a.bla	Modifies commut11 by putting edits between blocks into the last defined block. The edits in this block refer to fields in the other block by direct reference.
commu11b.bla	Modifies commut11 by putting edits between blocks into the last defined block, but this time using IMPORT parameters for the edits.
commut12.bla	commut11 but with included files bperson.inc and bdistanc.inc.
commut13.bla	Makes a table, with enumerated rows, out of commut12 and puts edits between blocks at the table level.
commut14.bla	A table, with arrayed rows, and shows how to put edits between blocks in a FOR-DO loop at the table level.
commut15.bla	Basic table definition. Also shows a table with unequal row lengths.
commut16.bla	Shows a table with holes.
ncs01.bla	Basic start to an elaborated example. Shows special administrative blocks and parallel status.
ncs02.bla	Adds hierarchical blocks and tables to ncs01. Up to 10 members of a household are interviewed.
ncs03.bla	Elaborates on ncs02 with additional instances of administrative blocks to handle individual appointments, nonresponse, and so on.
test14.bla	A small test instrument corresponding to commut14. This shows how use of parameters can make it easy to test even deeply nested blocks.
distance.bla	Example of a mini-data model used to develop and test for eventual use in a larger data model.
menujump.bla	Use of a menu at a field to determine which block to process at any time.
hh1.bla	First of five ways to build a household roster.
hh2.bla	Second of five ways to build a household roster.
hh3.bla	Third of five ways to build a household roster.
hh4.bla	Fourth of five ways to build a household roster.
hh5.bla	Fifth of five ways to build a household roster.
hh6.bla	Protecting a household grid.

5 Special Topics

Chapters 3 and 4 covered the basic Blaise® language and much more. This chapter covers features that are essential in some surveys but not needed in others. Topics include hierarchical coding, external files, lookup coding, Blaise® procedures, Dynamic Link Library (DLL), audit trail, multimedia language, question-by-question aids, and the layout section.

5.1 Hierarchical Coding

Coding means to assign a numeric code that corresponds to a description of a commodity, chemical, car, crop, or other kinds of items in a large list. Blaise supports three kinds of coding: hierarchical, alphabetical lookup, and trigram lookup. Some coding frames have a natural hierarchical nature and you can take advantage of the structure. Examples of hierarchies in coding frames include:

- Classification of commodities, occupations, and enterprises
- Make, model, and body style for automobiles
- Region, sub-region, and towns for place names

When you code hierarchically, you proceed from high-level classifications to low-level classifications. In doing so, you quickly narrow down the search, eliminating thousands of inappropriate low-level codes right from the start.

You can combine hierarchical coding with the lookup coding techniques covered below. For example, when coding automobiles, you can choose a make of car through a hierarchical mechanism. Chevrolet is a make of car. Since Chevrolet has many models, you can switch to lookup coding to search for the model name and body style. If you do so after choosing Chevrolet as the make name, the lookup coding mechanism searches for model names associated with Chevrolet only. When you switch to the lookup mechanism, all of the lookup search techniques (alphabetic, trigram, visual browsing) can be made available to the coder.

In Blaise, the hierarchical coding frame is implemented as a special kind of hierarchical enumerated type, called a *classification* type. You can store the

prepared classification type in a type library as documented in Chapter 3. You invoke the hierarchical coding with a *classify* method that is attached to the field to be coded.

5.1.1 Classification type

The hierarchical coding frame is implemented as a hierarchical enumerated type in the type section or type library. The syntax, formally covered in the *Reference Manual*, is best illustrated with an example showing enumerated tables.

```
LIBRARY FoodLib

TYPE
  Likes = CLASSIFICATION
  HIERARCHY
    Food = (
      Snacks = (
        Quiche,
        Pretzels,
        Peanuts),
      Drinks = (
        Juice,
        SoftDrink)),
    Fun = (
      Hardware = (
        Computer,
        Television,
        SoundSystem),
      Software = (
        Doom,
        Blaise,
        Lemmings))
  ENDCLASSIFICATION

ENDLIBRARY
```

Since the classification type is a series of nested enumerations, you can add the code you want to use and a description:

```
TYPE
  Likes = CLASSIFICATION
  HIERARCHY
    Food (1) "Anything you eat or drink" = (
      Snacks (1) "Food you eat between meals"= (
        Quiche (1) "Delicious cheese tart",

etc.
```

If the description is present, the interviewer sees only the description in the coding window. Otherwise the user will see the labels themselves.

Some coding frames have more complicated descriptions. From a hierarchical coding frame for automobile makes, models, and body styles, the labels may be quite long and complex, as shown in the following example:

```
TYPE
 Automobiles = CLASSIFICATION
 LEVELS
   Make, Model, Body_Style
 HIERARCHY
   American_Motors (1) = (
     Rambler_or_American (1) "AMER Rambler/American" = (
       _2dr_Sedan_or_HT_or_Coupe (2) "2dr Sedan/HT/Coupe",
       _4dr_Sedan_or_H (4) "4dr Sedan/HT",
       Station_Wagon (6) "Station Wagon",
 unknown_or_other_style (88) "unknown" ) ,
etc.
```

These three lines, taken from above, represent make, model, and body style levels of this coding frame:

```
American_Motors (1) = (
   AMER_Rambler_or_American (1) = (
     _2dr_Sedan_or_HT_or_Coupe (2),
```

An American Motors Rambler that is a two-door sedan, hard top, or coupe has code *1.1.2.* This example uses labels such as *AMER_Rambler_or_American* to represent the level being coded. Since these are identifiers, they must follow the rules of identifiers for enumerated type. Special characters such as a space, '-', or '/' are not allowed.

For this more complicated example, descriptive text in the classification type can make things clearer for the coder. The following is an excerpt from the same coding frame with shortened labels and descriptive text:

```
TYPE
 Automobiles = CLASSIFICATION
 LEVELS
   Make, Model, Body_Style
 HIERARCHY
   American_Motors (1) = (
     Rambler_or_American (1) "AMER Rambler/American" = (
       Auto_2dr (2) "2dr Sedan/HT/Coupe",
       Auto_4dr (4) "4dr Sedan/HT",
       Station_Wagon (6) "Station Wagon",
       Unknown (88) "unknown" ) ,
etc.
```

When coding, the coder would see *AMER Rambler/American* instead of *AMER_Rambler_or_American* as in the preceding example.

5.1.2 Building the classification type

The syntax illustrated above can be awkward to type by hand, particularly for large hierarchical coding frames. In any case, coding frames are rarely built up from scratch. If they exist in a different format, you can use Manipula programs to recast the frame into the needed syntax. For example, the coding frame above was created from a file with format:

```
1001 2 American Motors   AMER Rambler/American
       2dr Sedan/HT/Coupe       65
1001 4 American Motors   AMER Rambler/American
       4dr Sedan/HT             69
```

where the second and fourth lines are continuations of the lines immediately above.

Two Manipula set-ups that convert a source file to the two example classification types above `are carclass.man` and `carclas2.man`. Certain characters that appeared in the original file are not allowed within labels in the classification *type*. They are:

- A blank space
- A slash (/)
- A hyphen (-)
- A full stop or period (.)
- A comma (,)
- A plus sign (+)
- An ampersand (&)
- A quotation mark (")

In addition, a label cannot start with a number. The Manipula programs replace these characters with an appropriate substitute such as the underscore character, as shown in the following example:

```
HoldMakeName := REPLACE(HoldMakeName, '/', '_')
```

Here, the character / is replaced with the underscore character _. For this reason, the labels in the classification type should not have spaces but instead have underscore characters. An example (with codes deleted):

```
American_Motors
  AMER_Rambler_or_American
    _2dr_Sedan_or_HT_or_Coupe
```

In the second classification type example above, greater effort was taken to have more succinct labels. The corresponding labels are (with codes and descriptions deleted):

```
American_Motors
  Rambler_or_American
    Auto_2dr
```

Dynamic coding frames

Some coding frames are very stable and are changed infrequently. Others, however, can change as the survey progresses. If this is the case, you can give the coding frame the DYNAMIC attribute to turn off the type checking. This allows you to change the frame without changing the data definition. If you declare a coding frame to be dynamic, then you have to give a maximum length to the numeric code. In this example of coding automobiles, up to two digits are reserved for the make, three digits for the model, and three digits for the body style.

The levels have full stops between them, for example:

```
20.032.88
```

This frame has a maximum length of nine characters. To make it dynamic, the heading of the classification type would be:

```
TYPE
  Automobiles = CLASSIFICATION DYNAMIC[9]
```

Level names

You can give names to the levels of the hierarchy and use the level names in an IF condition. In the automobile coding example, the levels are named as:

```
TYPE
 Automobiles = CLASSIFICATION
 LEVELS
   Make, Model, Body_Style
```

Suppose a field named *Car* has type *Automobiles*. In the main data model, you can use the level name in a condition as shown:

```
IF Car.Make = Chevrolet THEN
```

You cannot use the level names directly in an assignment.

5.1.3 Classify method for coding a field

So far, we have only discussed how to create the coding frame in Blaise as a classification type. You still have to declare a field in terms of the type and give it a classify method. In the automobile example above, the classification type created had the name *Automobiles*. In order to use this classification type you have to define a field in terms of it. For example:

```
FIELDS
  AllCodes "Code the car." : Automobiles
```

In the RULES section, you use the CLASSIFY instruction to invoke the interactive hierarchical coding.

```
RULES
  AllCodes.CLASSIFY
```

When the user (or coder) arrives at the field *AllCodes*, the coding mechanism is activated by pressing the Insert key or by entering the starting value of a code. Once in the hierarchical coding mechanism, the coder can easily proceed down the hierarchy with page and arrow keys until the proper code is obtained.

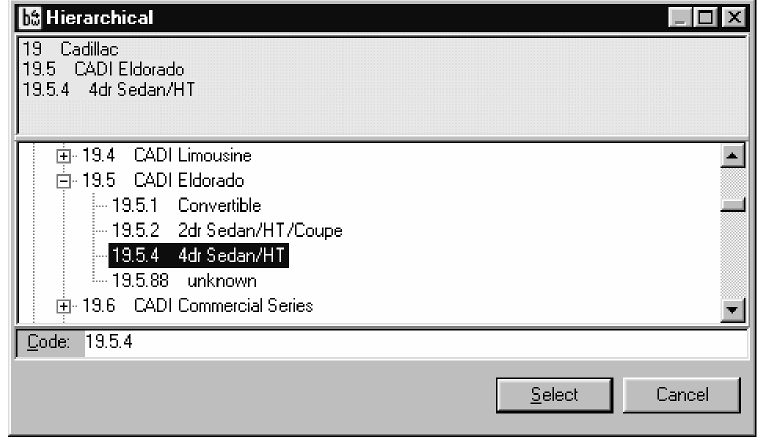
As an example, the data model `carcodes.bla`, its classification types, and related external files are found in the subfolder `\Doc\Chapter5\Classify`. The text file `read.me` gives complete instructions on how to prepare the data models and libraries.

When you run `carcodes.bla`, enter a *2* in the field *Methods* on the first page. You will arrive at the page that demonstrates hierarchical coding. In the field *AllCodes2,* press Insert and the following coding dialog box appears:

Figure 5-1: Hierarchical coding dialog box

The coding dialog box is in the form of a tree. By expanding and collapsing the tree (using the mouse or arrow keys), you can choose any of the makes of cars. Expand the tree at *Cadillac* and the tree unfolds to a lower level of the hierarchy for *model*. Expand *CADI Eldorado* and the lowest level of this hierarchy, body style, appears.

Figure 5-2: Coding dialog box



Select *4dr Sedan/HT*, then press the Enter key. Code *19.5.4* appears in the field *AllCodes2*. Press Enter again and the cursor will move to the field *Confirm2*. The

fields *ShowString*, *LongName2*, and *Confirm2* are imputed from an external file and give the interviewer a chance to confirm that the correct code was chosen. (External files are discussed later in this chapter.)

Selective use of the coding mechanism

You can invoke the coder under certain circumstances and not others by using the IF-ELSEIF or IF-ELSE construct:

```
IF CAPI THEN
  AllCodes.CLASSIFY
ELSE
  AllCodes.ASK
ENDIF
```

Coding from an open question

Open questions in Blaise are designed to collect verbatim responses from participants during the survey. If you later want to code this information, you can have a subject matter expert code the responses when the form arrives at the home office. In the DEP, you cannot have the open-question window and the coding window visible at the same time. But you can read in the response of the open question into the coding question as shown:

```
AllCodes "Code the car. @/ The car description =
         @Y^OpenCar@Y" : Automobiles
```

You can control the size and location of the coding window. You can set the coding window for the bottom half of the computer screen so that the text from the open question appears in the top half of the screen.

As an example, in the data model `carcodes.bla`, enter *5* in the *Method* field on the first screen and you will see an example of coding from an open question. When you enter a text string such as *'I drive a Cadillac Eldorado'* in the *OpenCar* field, the text for the field *AllCodes4* displays this text string for the benefit of the person doing the coding later.

5.1.4 Using the code later in the data model

When a code is entered you often want to do two things:

- Verify that the code is correct.
- Bring some related data into the data model.

Verifying a hierarchical code

To verify that a code is correctly entered, display the labels of the code in the text of a confirmation question. For example, where the field *Car* is of classification type *Automobiles*:

```
FIELDS
  Confirm "Let me confirm the make and name of the car.
  They are ^Car" : (Yes, No)
```

An edit could be written where an answer of *No* would prevent the coder from moving on until the code is correct.

Accessing external data based on a hierarchical code

Hierarchical coding is implemented through a type, not an external file as lookups are. When you arrive at a code strictly from a hierarchical search, you do not automatically have access to other data associated with the hierarchical code as you may have in lookups. You can associate the hierarchical code with data from an external data model using the SEARCH and READ methods. The external data model must have a primary key of the same classification type as the code field. Then you can use the code field (in this case, *Car*) as a search parameter from an external file. For example:

```
CarList.SEARCH(Car)
```

Use a READ statement to read the contents of the record into the external field *CarList*. For example, if you have an external file with some rating for a model of a car, you could bring information in from the external file for further edit checks. In the earlier example, where code *19.5.4* was selected, an external file filled in the fields *LongName2* and *BodyStyle2*.

Converting between code and string

If you need to manipulate the value of the code, use the function CLASSTOSTR. This will convert the code to a string value. Then you can use other string functions to do what you need to do. To convert a string to a code, use the function STRTOCLASS.

5.2 Retrieving Information from External Files

External files can hold information that changes over time or between regions. If the external file holds a large number of records, you can access the one you need very quickly based on data already collected during the interview. For some applications, appropriate use of external files will cut down on the maintenance of the main data model. The uses of external files are many. Examples include:

- Verifying that codes input during an interview are valid.
- Comparing the answers for different periods in a longitudinal survey. Therefore, you need a facility to read information from the previous period.
- There is information associated with the sampling frame that is needed within the form. For example, expansion factors may depend on stratum code and these expansion factors are multipliers in some edits. This frame sampling information is held in an external data set.
- After coding a commodity, information related to that commodity must now be brought into the data set. This could be for further computations and edit checks or because the summary system requires the newly collected data to be with previous data.
- Running an instrument that must behave in different ways for different regions. You can use an external file to hold specifications for each region. Then the main data model will adjust routes, edit limits, and even question text to correspond to the specification for the region where it is being used.

5.2.1 External file requirements

There are a few requirements of an external file. For the externally held data, you need:

- An external Blaise data model with a primary key to describe the external data file.
- An external ASCII data file, a Blaise data file, or a relational database using the Blaise OleDB interface. For an ASCII text data file the Blaise data model must describe the fields exactly —their position, width and data type. A Blaise data file may be built by converting the ASCII data using Manipula. Accessing a relational database such as Oracle®, Microsoft® SQL Server™, or Microsoft Access® is possible when the Blaise Component Pack is installed. More information on the Blaise OleDB interface can be found in section 2.2.9 and in the Control Centre Help under Reference Manual > BOI > BOI Externals.

Note that there is a distinction between the external data model and the external data file. In the main data model, you need to refer to both the external data model and the external data file. For the main data model you need:

- A USES section at the start of the main data model to name the external data model.
- An EXTERNALS section in the block where the external file is read. The EXTERNALS section names the external data file.

In the same block as the EXTERNALS section, you locate and read a data record in the external file. To do this you need:

- The SEARCH method to locate a record in the external file.
- The READ method to read the selected record in the external data file.

5.2.2 The external data model and data file

Reading an external data file in standard ASCII format has advantages. The translation step to a Blaise data file is eliminated, it is often easier to update or change the data file, and for small datasets there is likely to be little impact on performance. For larger datasets (based on the number of records and/or the length of the records) some performance impact can occur during the initialisation step.

Using a Blaise data file for external files is likely to be of value when the size of the dataset is large or when it does not change frequently, or if the data have been collected using Blaise.

If the external information is in a different format, bringing it into the Blaise format means specifying an external Blaise data model. Usually this is very simple.

We will start with a short example from the National Commuter Survey. You want to describe the respondent's commuting trips as a sequence of modes of transport. For example, a respondent may walk 1 kilometre to the metro station, take the subway to another station, catch a bus there, and finally walk another 1/2 kilometre to his office. His sequence of modes would be *walk, subway, bus,* and *walk.* You do not know in advance what the sequence of modes will be. However, for each mode of transport you have a particular phrase for proper wording of questions and edit limits for average speed. An ASCII file of this external information might look like this (`from modelist.asc`):

```
 1take the public bus           10 80
 2take the private bus          10 80
 3take the tram or trolley      10 80
 4subway, metro, or light rail  15 65
 5take the train                25 90
 6take your car by yourself     25 85
 7take the car or van pool      25 85
 8take your motorcycle          25 95
 9take your bicycle             15 40
10walk                           5 15
11take other means of transport  0150
```

A description of the external file might be written as:

Columns	Length	Description
1 - 2	2	Mode number {unique id}
3 – 32	30	Proper question phrase
33 – 34	2	Lower edit limit km/hour
35 – 37	3	Upper edit limit km/hour

A Blaise data model corresponding to this description of the external file could be:

```
DATAMODEL ModeList
  PRIMARY
    ModeNumber
  FIELDS
    ModeNumber : 1..11
    ProperPhrase : STRING[30]
    Lower : 0..50
    Upper : 1..200
ENDMODEL
```

A Manipula program to bring this ASCII file into a Blaise data set would be (`modelist.man`):

```
USES
  Modelist

INPUTFILE
  InFile : Modelist ('modelist.asc', ASCII)

OUTPUTFILE
  OutFile : Modelist ('ModeList', BLAISE)
```

Manipula is covered in Chapters 7 and 8, including how to check and run this set-up.

One can also use ASCII files with fields separated by a comma or other delimiter, with data like the following:

```
1,"take the public bus",10,80
2,"take the private bus",10,80
...
11,"take other means of transport",0,150
```

In the Manipula specification of the INPUTFILE one uses the SETTINGS section to state the delimiter and field separator characters.

```
USES
  Modelist

INPUTFILE
  InFile : Modelist ('modelist.asc', ASCII)
  SETTINGS Delimiter='"' Separator=','
OUTPUTFILE
  OutFile : Modelist ('ModeList', BLAISE)
```

Use of an external data model to state specifications

If you do not already have an ASCII file of the external information, you can use a Blaise data model to record the information. The data model above is adequate for this but you can take the idea further. With a few enhancements, you could give the data model to a subject matter specialist to fill in the question wording and edit limits. You could also put a few edits in the external data model itself, such as ensuring that the field *Lower* is always less than *Upper*. An enhanced data model is `modelist.bla` found in `\Doc\Chapter5\External`.

The main data model used in this example is `commut17.bla` in `\Doc\External\Chapter5`. This data model has a table for modes of transport where it is unknown ahead of time which mode of transport will be mentioned by the respondent at any time. It is necessary to read an external file to retrieve a question phrase and edit limits appropriate to the mode stated.

5.2.3 Referring to the external data model and data file

From the main data model you indicate which external data model and data file to read. You name the external data model in the USES section and the external data file in the EXTERNALS section. The external data model and data file are treated separately because the data model specification may apply to two or more data files with the same description.

Uses section

State the name of the external data model in the RULES section at the beginning of the main data model. Associate a *MetaInformation* identifier (*ModeModel* below) with the data model name. It is the *MetaInformation* identifier that is used further in the main data model. In the following example, the external data model name is *ModeList*.

```
DATAMODEL Comute17 "National Commuter Survey, ex 17."
 . . .
USES
  ModeModel 'ModeList'
```

If the external metadata file is held in another directory you can use a path:

```
DATAMODEL Comute17 "National Commuter Survey, ex 17."
 . . .
USES
  ModeModel 'c:\TranInfo\ModeList'
```

The system will look for the external metadata file `modelist.bmi` and read the external data descriptions from it. That is, all the information about the external data model, such as fields, block names, and data definition, is available from the metadata file. It uses the identifier *ModeModel* to refer to this metadata.

You can omit the file specification if the name of the identifier is the same as the name of the metadata file. One *MetaInformation* identifier can refer to two or more external data files as long as they all have the same data definition.

Externals section

Name the external Blaise data file in a RULES section. Associate an *external field* (*ModeFile* below) with the *MetaInformation* identifier from the RULES section (*ModeModel* below) and the external data file (*ModeList* below). For our example this section looks like the following if a Blaise data file is used:

```
BLOCK BDistance  {where the external data are used}
 . . .
EXTERNALS
  ModeFile : ModeModel('ModeList')
```

If an ASCII data file is used then the section looks like:

```
BLOCK BDistance  {where the external data are used}
  . . .
EXTERNALS
  ModeFile : ModeModel('ModeList.asc', ASCII)
```

Again, you can refer to a path if necessary:

```
EXTERNALS
  ModeFile : ModeModel ('c:\TranInfo\ModeList.asc', ASCII)
  SETTINGS Delimiter='"' Separator=','
```

If you do not include a data file name, the same name as the metadata file name will be used (this is the default).

The external field identifier (in the above example, *ModeFile*) refers to one external data record. Once the external file is read, the external field will hold one external data record with all fields in that record. In our example, these fields include *ModeFile.ModeNumber*, *ModeFile.ProperPhrase*, *ModeFile.Lower*, and *ModeFile.Upper*. You can refer to the contents of these fields in the RULES after the external SEARCH and READ are accomplished.

Define externals at the lowest level

The RULES section with the SEARCH and READ methods must be located in the block where the file searching and reading are done. This should be the block where the external data are to be used. In other words, declare the RULES section at the lowest level possible. This is especially valuable in large arrays where you might have to read many external files.

Take the example of an arrayed table with many rows. The best way to handle this is to read the external file from within the row block definition. By doing this there will be no parameter administration associated with the external file read, and instrument performance will be maintained. See Chapter 4 for a discussion of parameter administration and performance.

Restricting the number of external fields

If you are only interested in part of the data in the external data record, you can restrict the information that is read from the file. To do this, sum up the fields of interest in the specification of the external field. For example, if your external data model consists of two block fields, one with identification information and

one with demographic information, and you are only interested in the identification, you can specify:

```
EXTERNALS
  TownData: TownName
  ('C:\TownDir\Towns', Identification)
```

When the system processes a record in the external data file, only the fields in the block *Identification* will be read. Note that you cannot refer to fields that are not mentioned in the specification. By default all fields are read and available.

You can specify more than one field to be read. If you have several fields of interest you can sum them up in the specification. For instance:

```
EXTERNALS
  TownData: TownName
    ('C:\TownDir\Towns', Identification, Income)
```

5.2.4 Accessing the external data with file methods

So far you have set up references to the external data model and data file with the USES and EXTERNALS sections. Now you must use file methods to actually read the external data. At this point you only need to use the external field from the EXTERNALS section. The file methods that you can use are SEARCH, READ, OPEN, and RESULT.

To activate a file method, specify the external field name followed by a dot and the desired method. For the READ method that would be:

```
ModeFile.READ
```

To read data, the SEARCH and READ methods are used together.

SEARCH method

The SEARCH method searches the external data file for a record based on the external file primary key. SEARCH takes as many parameters as there are fields in the primary key of the external file. In the example there is one field in the external primary key. The search instruction is:

```
ModeFile.SEARCH(Mode)
```

Mode refers to a field, auxfield, parameter, or local in the block in the main data model. In our example, *Mode* is a field that the interviewer enters. When *Mode* gets a value, the SEARCH method tries to locate the primary key with the same value in the external file and returns the result of the attempt: successful or not successful. It also prepares the system for reading the data record.

Usually you use the SEARCH method with some IF conditions.

```
RULES
  Mode
  IF ModeFile.SEARCH(Mode) THEN
    {more code}
  ENDIF
```

In the example above, if the SEARCH is successful, then more code will be executed. In this example, there is no 'way out' for the interviewer (there is no ELSE or ELSEIF before the ENDIF where the SEARCH is done). It is up to the developer to make sure that there are records in the external file for each possible value of *Mode*.

Since the SEARCH method returns the result of the search, you can give the user an alternative if an external record cannot be found. For example, in a different context where we want to impute a town name based on a town code, we allow the interviewer to type in a name if one cannot be found.

```
IF TownData.SEARCH(TownCode) THEN
  TownData.READ
  TownName := TownData.TownName
  TownName.SHOW
ELSE
  TownName.ASK
ENDIF
```

If the search is not successful, the town name will be asked and the interviewer can type in a town name.

There may be more than one primary key field in the external data file. In that case you must specify a value for all the keys, separated by commas. For example, if the external data file has two key fields, one for region and one for mode of transport, the SEARCH might look like:

```
IF ModeFile.SEARCH(RegionNum, Mode) THEN
```

You can use constants, fields, auxfields, locals, function results, and expressions as parameters of the SEARCH method. The only restriction is that the values should be of the proper type.

You can use the contents of the external file to perform a direct edit check. In the example below, we want to ensure that the number of modes of transport corresponds to an entry in the external file. If it does not, this indicates a data entry error. A direct check with a search would look like:

```
ModeFile.SEARCH(Mode)
  "Mode of transport not found, re-enter the number."
```

Here you are not actually reading the external data record. You are merely checking to see if it exists. If it does not exist, then there is an error.

READ method

Data can be read with the READ method after a successful search:

```
RULES
  Mode
  IF Mode <> EMPTY THEN
    IF ModeFile.SEARCH(Mode) THEN
      ModeFile.READ
    ENDIF
    {more code}
  ENDIF
```

In this example, one external data record will be read. The information in the record can be addressed with dot notation. Since the record represented by *ModeFile* has fields *ProperPhrase, Lower,* and *Upper*, refer to these fields with *ModeFile.ProperPhrase, ModeFile.Lower,* and *ModeFile.Upper*. For example:

```
FIELDS
  Distance "^Respondent, what is the distance you travel when you
            ^ModeFile.ProperPhrase to work?" : 0.0..200.0

SIGNAL
  AvgTime > ModeFile.Lower
```

In the first part of the example, *ModeFile.ProperPhrase* is used to modify question text in the main data model. In the second part, the external field name *ModeFile.Lower* is used to state an edit limit.

RESULT method

The RESULT method supplies information about the result of the last SEARCH or READ operation. RESULT returns the value zero if the operation was successful, and an error code if it was not. For instance:

```
TownData.READ
IF TownData.RESULT = 0 THEN
  TownName := TownData.Name
ELSE
  TownName.ASK
ENDIF
```

If you use the SEARCH method, there is usually no need to check the result.

OPEN method

You can use the OPEN method to switch to another data file. OPEN takes one string parameter, the name of the data file. The string may include a drive letter and a directory:

```
ModeFile.OPEN('C:\TranList\ModeLst2')
```

You can also use a string type expression, as in:

```
TownData.OPEN(CurrentDir + '\TownDir\' + TownFileName)
```

where *CurrentDir*, *TownDir*, and *TownFileName* are locals, fields, or auxfields and are computed with the current directory name, the TownDir directory, and the town file name before the above instruction is executed.

Longitudinal surveys

For longitudinal or panel surveys, there are two ways that data are typically collected and edited. The first is where the user is allowed to collect and edit current data only. The second is where the user must be able to update previous data.

In the first case, you can develop your instruments along the lines described above. The external Blaise data models are those that you used for data collection or editing in previous periods. In this situation, you have access to the information from the previous data models but cannot change it.

In the second case, when you want to be able to edit previously collected data, you might consider building one data model that can handle all survey periods. You can use routing and conditional statements to control access to the current data. For example, if you are in period three, you can allow access to periods one, two, and three, but not to period four.

Large external files

Very large files can be read. The primary key of the external data file is an index to the data records. This index will ensure that the required record is read very quickly. You often use large external data files in association with some coding scheme. There, it is not unusual to have tens of thousands of records in the external file. Blaise can handle these situations easily.

Many external files

You can have an unlimited number of external files from a main data model. When you have many external data models you must be sure that you have enough memory to handle all the data models and data files.

Large external data model

You can refer to a large external data model in the RULES section. However, you must make sure that you have enough memory to hold the main data model (`.bmi` file and other information), the external data model (`.bmi` file), one or two data records for the main data model, and one data record for the external data file. If you do not have enough memory, you can extract information from the large external data file into a smaller external data file with a smaller external data model description.

5.3 Lookups

Lookups are used to browse external files for information or for coding. You look up information in the external file by scrolling with arrow and page keys, using an alphabetic search, or using a powerful text locating method called trigram search. Lookups can be used in combination with hierarchical coding.

Lookups differ from external file searching and reading covered above. With external file searching and reading, you give or calculate search criteria within the main instrument to find a single record in the external file. The user may not even know that information was obtained from an external source. With lookups, a

window opens that displays the external file and the user is given several ways to find the proper record.

With external file lookups you can:

- Code commodities, automobile names, place names, job descriptions, and similar lists through text-string searches.
- Quickly locate the needed record even in files of thousands of records.
- Give the interviewer information. This might be to answer common respondent questions or to look up descriptions. The lookup is always available if implemented in a parallel block. (Another way to give the interviewer information is through question-by-question aids covered later in this chapter.)

Lookups can be used in place of or in combination with hierarchical coding.

5.3.1 External lookup file

The external lookup file must have at least one key field. It will almost always have at least one descriptive text field to aid the user in locating records. The lookup file may have a primary key, one or more secondary keys, or both.

- If the lookup file has only a primary key, then the lookup mechanism offers a window on the lookup file in the order of the primary key. Usually the primary key is a numeric identifier and does not offer much help to the user who is trying to find a particular record.
- The lookup is most powerful if the lookup keys are text descriptions of the records being sought. These are usually secondary keys. For example, the make and model names of a car are good descriptions of the records being sought and make good search fields.
- If there is a primary key together with secondary keys, then the secondary keys but not the primary key are offered as lookup keys. You might still find it advantageous to declare a primary key if you want to offer the external lookup file as a regular external file for the search and read methods. This is a good idea if you combine hierarchical coding with a lookup.

An example of an external data model used in lookups is `\Doc\Lookup\Chapter5\Lookup\carlist.bla`.

```
DATAMODEL CarList {Used both as an external data model for the main data
                   model LOOKCARS.BLA and for the Manipula program which
                   populates this Blaise data base.}

  SECONDARY
    Long_Name = LongName
    TrigramMakeAndModel = LongName (TRIGRAM)

  FIELDS
    MakeName  : STRING[18]
    LongName  : STRING[34]
    BodyStyle : STRING[25]
    ModYear   : INTEGER[2]
    AllCodes  : STRING[7]

ENDMODEL
```

Once you prepare this data model you can populate it using the Manipula program `carlist.man`. This will place 1,276 records in the external file. After data are read into this external Blaise data model, prepare the main data model `lookcars.bla`. With these two data models you can experiment with lookups, especially with the different kinds of keys in the external model `carlist.bla`.

From the main data model `lookcars.bla` we want to record the make, model, and body style of the respondent's automobile. A unique code is held in *AllCodes*, and this is the number we need in our main data model. However, *AllCodes* is not a good search field because it contains just code numbers. One of the fields in the external data model, *LongName*, contains both a short make name and a model name. For example:

```
CHRY New Yorker/E-Class
FORD Mustang/Mustang II
```

Since *LongName* would make a good search field, we will use it to find the proper code in *AllCodes*. This can be with either an alphabetic search or a trigram search.

5.3.2 Keys in the external lookup file

The way the keys are defined in the external data model determines how the user can search the lookup file. Since there is no primary key in this external lookup, only the secondary keys are offered as lookup keys to the user. A normal secondary key (*Long_Name* in this example) is used for the alphabetic search. A secondary key that is given the TRIGRAM designation (*TrigramMakeAndModel* in this example) allows a powerful search based on text strings. In `carlist`, since

the secondary key *Long_Name* is listed first, it is offered to the user first when the lookup is invoked.

Alphabetic search

In an alphabetic search, the user starts to type in the name of the description as it is spelled. For example, in order to code the Ford Mustang above, she starts typing in *Ford*. As she types, the lookup file scrolls to the correct area of the external file. Just typing in the letter *F* will get her to the entries that start with *F*. Usually typing in the first few letters will get the user close to the correct external record. From there she can use the arrow keys to put the cursor on the right code.

The way this external file is set up, *LongName* will usually be an adequate alphabetic key as long as the respondent knows the make of the car. In order to use *LongName* for an alphabetic search, declare it as a normal secondary key. For example:

```
SECONDARY
  Long_Name = LongName
```

Trigram search

The trigram search is based on three-character text strings found anywhere in the search field. For example, the word *Blaise* consists of the trigrams *#BL, BLA, LAI, AIS, ISE,* and *SE#,* where # represents word boundaries. An algorithm inspects the trigrams based on what the coder is typing. As she types more, the number of qualifying records is reduced.

For example, out of 1,276 entries in the external file `carlist`, the search is narrowed to 11 entries when you type in the model name *nova*. The word being searched for does not need to be at the start of the search field, and in fact, it usually is not. After narrowing the search, the arrow keys or the mouse can again be used to choose the final code. The use of arrow keys often will be necessary for this example because part of what we are coding, the body style, is not included in the search field *LongName*.

To implement the trigram method, give the secondary key the TRIGRAM designation. For example:

```
SECONDARY
  TrigramMakeAndModel = LongName (TRIGRAM)
```

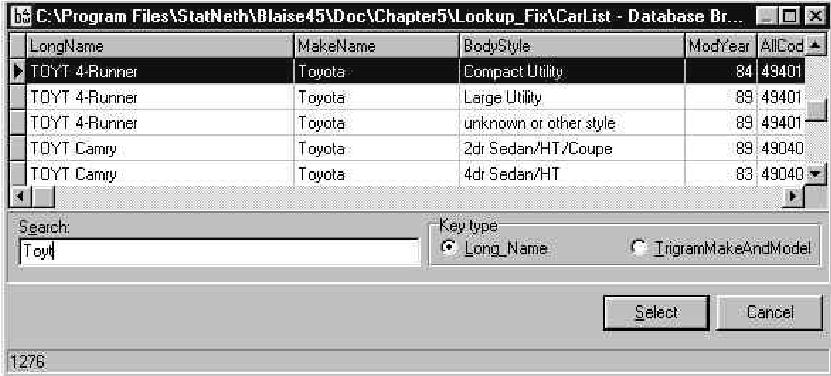
Note that in order to use trigram searching the external data file must be a Blaise database with a trigram secondary key. ASCII external files and OLEDB files cannot be used.

Combination of keys

You can use combinations of keys. For example, you can use one field for a trigram search and another for an alphabetic search. You can even declare one external field to be both a trigram field and an alphabetic field, as is done in `carcodes.bla`.

If you use combinations of keys, you can switch between keys in the lookup window. First select the lookup key you want to use from the *Key Type* box in the lookup window. Then type the search string in the *Search* box. In our example, we have searched on *TOYT:*

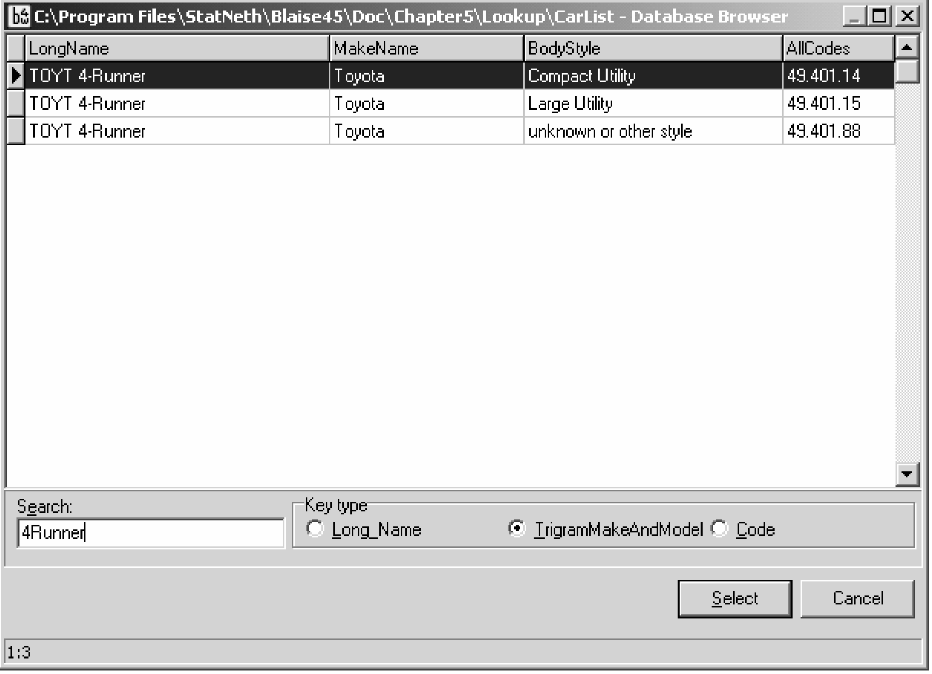
Figure 5-3: Alphabetic lookup coding dialog box



In the preceding example, since *4-Runner* is at the top of the Toyota list, the code is easily found using the alphabetic string.

A more robust method is to use the trigram search. You could search on *4-Runner*, and find it in the list. In the following example, even though the string *4Runner* was typed without the hyphen, the system still takes the user to the code because the search string is close enough to the target string *4-Runner*.

Figure 5-4: Trigram lookup coding dialog box



5.3.3 Declaring the external file lookup from the main data model

You need a USES section and an EXTERNALS section to identify the external lookup file. For example:

```
USES
  CarList
```

In the externals section you should define a reference:

```
EXTERNALS
  CarList : CarList
```

Lookup as an information source

In the RULES you invoke the method LOOKUP by adding this key word to the external file name. The following will merely offer the lookup as an information source:

```
RULES
  CarList.LOOKUP
```

Lookup for coding

If you are using the lookup file to code a value, then you need a field in the main data model that can accept the code:

```
{In main data model.}
FIELDS
  AllCodes "Code the car. " : Automobiles
```

The type of field in the main data model must be the same as the type of field that holds the code in the lookup file. For example, if the type of field in the lookup data model is a string, then the field in the main data model must also be a string.

Piping symbol

To use the lookup as a coding mechanism, in the rules you need to associate a field in the main data model with one in the external data model. This is done with the piping symbol '|'.

```
RULES
  AllCodes | CarList.LOOKUP.AllCodes
```

When the user arrives at the field *AllCodes* and starts typing, the lookup table will appear. She can then use various means to arrive at the proper code.

The piping symbol '|' is an alternative way to put a value into a field. In this case it is used to obtain a value from the external lookup. It is also used in the classify method below. Piping is further documented in the *Reference Manual* under *piping*.

5.3.4 Accessing related data in the lookup record

Lookups are a way of searching data records in an external file. Once a record is chosen, then all the data in the external record are held in the external field. In this example, the external field is *CarList*. Through *CarList* the fields of the chosen record are available to the main data model. You compute external values into the main data model as shown:

```
IF CarList.LongName <> EMPTY THEN
  LongName := CarList.LongName
  BodyStyle := CarList.BodyStyle
ENDIF
LongName.SHOW
BodyStyle.SHOW
```

This example brings in external data to the main data model for the coder to confirm whether the correct code was entered. The fields *LongName* and *BodyStyle* are displayed on the screen. The interviewer can see if he has coded correctly. The IF condition is necessary in case you bring up the record after storing it. Without the IF condition, an EMPTY would be computed into the fields *LongName* and *BodyStyle* upon recalling the form into memory after storing it. This is because the external field *CarList* would be EMPTY the second time around, as it is filled in from the external *lookup* file only when the lookup is manually activated and a record is chosen.

5.3.5 Giving the lookup a starting value

You can give a starting value for a search in the lookup file. This is best illustrated by an example:

```
FIELDS
  SVLook "Starting characters of the description. "
         : STRING[10] {for alphabetic search}
  SVTriGram "Any character string in the description"
         : STRING[10] {for trigram search}
```

In the RULES you feed the value of *SVLook* into the search key as in the following examples. Remember that the lookup file has secondary keys *Alfa* and *Tri*. If you want to feed a value for an alphabetic search:

```
AllCodes | CarList.LOOKUP(Long_Name:= (SVLook)).AllCodes
```

If you want to feed a value for a trigram search:

```
AllCodes | CarList.LOOKUP(TrigramMakeAndModel := (SVTrigram)).AllCodes
```

If you want to feed a value for a trigram and alphabetic search with the alphabetic being the first used:

```
AllCodes | CarList.LOOKUP(Long_Name := (SVLook),
           TrigramMakeAndModel:= (SVTrigram)).AllCodes
```

Feed values into the search based on information gathered previously in the instrument. For example, if you already know the make of the car from another question, there is no sense in making the coder enter that piece of information all over again. You can, however, back up in the coding mechanism and enter a different high-level code.

5.3.6 Using hierarchical coding and lookup together

For complicated or large coding frames, the combined use of the hierarchical and lookup coding methods is very powerful and elegant. The coder can use hierarchical coding to code the first levels and then switch to lookup coding to complete the job. When switching from hierarchical to lookup, only the entries in the chosen class are available to the coder. In other words, you can use the hierarchical coding to narrow down the search before switching to lookup coding.

Linking hierarchical and lookup coding

The link between hierarchical and lookup coding is based on the common use of the same classification type. In the example of automobile coding, there is a classification type called *Automobiles*. In the main data model, the field to be coded is defined in terms of *Automobiles*. In the lookup data model, you also define the field, which holds the code in terms of the classification type *Automobiles*. Then you use piping in the rules of the main data model to link the two fields together.

```
AllCodes.CLASSIFY | CarList.LOOKUP.AllCodes
```

In this case, if the coder switches from hierarchical coding to lookup coding, the code value from lookup coding will be piped into the field *AllCodes* in the main data model.

Open nature of combining the coding methods

When the hierarchical and lookup coding methods are combined, you are giving the coder an option. In a particular case, she may use only the hierarchical mechanism, only the lookup mechanism, or both.

If you wish to bring other external information into the data model where a combination of hierarchical and lookup coding is used, it is best not to rely on an

external computation that assumes the lookup mechanism was invoked, because the coder may have coded the entry entirely through the hierarchical method. In this case it is better to use the SEARCH and READ methods to ensure that you can access the external information regardless of the coding method the coder used.

5.4 Blaise Procedures

Blaise *procedures* can execute repetitive tasks within or between applications. With them you can:

- Invoke a common form of an edit check to many different fields.
- Add edits to an instrument after it is in production without changing the data definition.
- Encode complex tasks of several or many lines of code and apply them to different situations.
- Ask for information from a user without storing it.
- Call an *alien procedure* to do something. (Alien procedures are explained later in this chapter.)

The structure of a Blaise procedure is like that of a block except that the key words PROCEDURE and ENDPROCEDURE are used instead of BLOCK and ENDBLOCK. You can use all language elements including fields, auxfields, locals, parameters, computations, arrays, checks, and signals.

A Blaise procedure is used with explicitly declared parameters to pass data back and forth with the main application. None of the procedure's data are stored. This means you can use fields within a Blaise procedure and not take up space in the Blaise data set. Checks or signals that are invoked from a Blaise procedure can influence the cleanliness status (dirty, suspect, clean) of the application, but the result of each check or signal in the Blaise procedure is not stored. Consequently, you can use a procedure to add checks or signals to your application after it is already in production. This will maintain data set compatibility between the new and old instruments (if you did not already reserve space for additional edit checks with the key word RESERVECHECK, as discussed in Chapter 3). A generalised Blaise procedure can be tested and maintained in its own file and used in different applications without modification.

The following is an example of a simple procedure returning a percentage:

```
PROCEDURE Percentage
  PARAMETERS
    Part, Total: REAL
    EXPORT Perc: INTEGER
  RULES
    Perc:= ROUND(Part / Total * 100)
ENDPROCEDURE
...
Income Rent
Percentage(Rent, Income, RentPerc)
RentPerc.SHOW
...
```

A procedure can be used to display information on screen and to retrieve input from the user.

```
PROCEDURE DispInfo
  PARAMETERS
    Info: String
  FIELDS
    Dinfo "^Info": String[1], EMPTY
ENDPROCEDURE
...
Name Address
DispInfo('The name and address just entered are: '+ Name + ' '+ Address)
...
```

Note that you can call the procedure directly; you do not have to specify a field first. A procedure does not link to data, so no information will be stored.

A more elaborate example of a procedure can be found in the file `M_of_20.prc` in the folder `\Doc\Procedure\Chapter5`. It is shown in the following example:

```
PROCEDURE M_of_20
  PARAMETERS
    IMPORT
      N, M  : INTEGER
    EXPORT
      Choices : STRING

  LOCALS
    L, K, J,
    PrevL : INTEGER

  FIELDS
    LetterN, LetterM, NumEligible : 0..20, EMPTY
    NumChosen : 0..20, EMPTY
    Choice : 1..20, EMPTY
    OrderedArray, RandomArray : ARRAY [1..20] OF 1..20, EMPTY

  RULES
    Choices := ''
    IF M > 0 AND N > 0 THEN
      LetterN := N
      LetterM := M
      FOR L := 1 TO LetterN DO
        OrderedArray[L] := L
      ENDDO
      NumEligible := N
      Choice := 1 + RANDOM(N)
      FOR K := 1 TO LetterM DO
       RandomArray[K] := OrderedArray[Choice]
        Choices := Choices + STR(RandomArray[K], 3)
        NumChosen := NumChosen + 1
        NumEligible := N - NumChosen
        FOR J := Choice TO NumEligible DO
          OrderedArray[J] := OrderedArray[J + 1]
        ENDDO
        OrderedArray[NumEligible + 1] := EMPTY
        Choice := 1 + RANDOM(NumEligible)
      ENDDO {K = 1 TO LetterM}
    ENDIF {M > 0 AND N > 0}
ENDPROCEDURE
```

This procedure returns *m* unique elements out of 20 in a random sampling without replacement scheme. This procedure is necessary because the Blaise RANDOM function returns only one value at a time, and two or more successive invocations of RANDOM will not ensure uniqueness.

The procedure above is quite complex. There are many explanatory comments in the file itself if you are interested in the details of the code. The above example illustrates that you can have very complex instructions in a procedure. Once programmed and tested, the procedure is available to other instruments without modification.

5.5 Dynamic Link Libraries

Blaise supports the use of Dynamic Link Libraries (DLLs) to perform a process or action that is not currently available in Blaise itself. By using DLLs, Blaise can perform many tasks that would not normally be possible.

A DLL can be considered an external subroutine that can be called by the Blaise Data Entry Program (DEP) or Manipula. The Blaise instrument passes information to the DLL, and the DLL acts on the information and passes the modified or new information back to the instrument.

The following are some uses of DLLs:

- You can read data from a serial port. For example, you could scan the bar codes of food products found on the shelves of a respondent. Another use would be to use electronic callipers to measure fruit or machine parts. For this you would use an *alien router*.
- For a self-interviewing application, you can use a totally different interface for the respondent than the one the interviewer uses. For this you would use an *alien router*.
- You can run a third party program for a specialised kind of coding. For this you would use an *alien procedure*.
- If you have some administrative survey data stored in a database system, you can read data from that package directly. For this you would use an *alien procedure*.

DLLs are a specialised technique that should be used only by a knowledgeable computer programmer. It is not for the novice. Valuable things can be accomplished with DLLs, but it is also possible to create problems with your application. You must thoroughly test the whole system with the DLLs before using them in production.

This section provides a brief overview of how DLLs are used in Blaise, information on how to create a DLL to be called from the DEP, and information on how to create a Blaise alien procedure and a Blaise alien router to call a specified DLL procedure.

This section only covers the very basics. Examples distributed with the Blaise system will offer much more detail. All files and examples mentioned in this chapter are available for review and are listed at the end of this section.

Also, a more in depth version of the Blaise DLL documentation is distributed with the Blaise system.

5.5.1 Two types of alien DLL reference

Blaise uses the key word ALIEN to refer to a DLL. There are two broad types of alien references. Those that perform calculations are called *alien procedures* and those that ask questions are called *alien routers*.

Alien procedure

An alien procedure is executed every time it is invoked in the RULES section of the data model when run in the DEP, or in the MANIPULATE section of Manipula.

The structure of an alien procedure is similar to a Blaise or Manipula procedure, with the exception that it includes the ALIEN key word. This enables the alien procedure to call a specified DLL procedure and pass parameters to it.

An alien procedure can only access and modify parameters passed to the procedure.

Alien router

An alien router is executed whenever the focus in the DEP (that is, the cursor) is placed on a field located within a Blaise block where an alien router statement is defined.

The structure of an alien router is similar to a Blaise block with the exception that it includes the ALIENROUTER key word. This enables the Blaise block to call a specified DLL procedure and pass a block of information to it. The block of information is comprehensive and detailed. The information includes every field declared in the Blaise block, as well as parameters passed to the block. Alien routers are much more powerful than alien procedures and are normally used to ask a question.

An alien router can access and modify parameters passed to the block, as well as all fields included within the related Blaise block.

5.5.2 Delphi™ DLLs and other DLLs

DLLs must be developed using Borland® Delphi™ (version 2 or later) because Blaise only delivers the DLL interface in the form of a Delphi unit. The Delphi

DLL itself can communicate with DLLs that are developed using a different development environment, such as C++. Thus to call a third party system written in C++, you would construct a small Delphi DLL to call the C++ DLL.

5.5.3 Delphi™ DLL procedure called by a Blaise DEP alien procedure

A Blaise alien procedure executes a Delphi DLL procedure every time the alien procedure is invoked in the RULES section. DLL procedures executed by an alien procedure can only access and modify parameters passed to the Blaise alien procedure.

5.5.4 Delphi™ DLL procedure called by a Blaise DEP alien router

A Blaise alien router executes a DLL procedure whenever the focus of the DEP (the cursor) is placed on a field located within a Blaise block where an alien router statement is defined.

5.6 Audit Trail

An audit trail in Blaise is a record of field values and movements in the instrument. Because Blaise is a Windows® system in which a mouse or other pointing device can be used, the audit trail must keep track of the position of the cursor resulting from the use of the pointing device as well as movements from the keyboard keys. It records the values of each field as the cursor enters and then again when the cursor leaves the field. In addition the action that brought the cursor to the field and then action related to exiting the field is captured. The audit trail also records when certain function keys are used, such as *Show All Remarks* or *Help*.

Audit trail uses

An audit trail has many uses in computer assisted surveys.

- It can be used for methodological research, in usability testing, and other efforts to understand how the user—the interviewer, editor, or respondent in a self-administered application—interacts with the instrument.
- An audit trail can tell you how interviewers are using the system, even in remote Computer Assisted Personal Interviewing applications. For example, if one wanted to learn how well a complex series of questions are functioning, an examination of audit trails could reveal patterns of backing up and

correcting answers, the use of help or other special functions, or the time spent on different items.

- An audit trail can help in the testing and debugging of the Blaise code. Complex flow through an instrument can be checked. Reported problems from software testers can be examined with much greater detail and precision rather than relying on problem reports.
- If there is a problem with the data files that the Hospital utility cannot fix, the audit trail can offer a means of data recovery.

The most compelling reason for using an audit trail is quality control. Audit trails provide an exact, in depth record of virtually everything that happens during a Blaise interview or editing session. Every instance where data in a Blaise instrument may be entered or changed by a user is recorded precisely.

Audit trail in Blaise

An audit trail is implemented through an external audit trail DLL. To implement the audit trail DLL for a survey, you need to:

- Supply an audit trail DLL.
- Tell the system where the audit trail DLL can be found.
- In the audit trail DLL, give the name of the audit trail and its location.

Note that there is a distinction between the audit trail DLL and the audit trail itself. The audit trail DLL enables an audit trail to be written. The audit trail itself records the user actions.

Since the audit trail is implemented by means of an external audit trail DLL, it is possible for you to create your own custom DLL and change how you format it or what information to record.

5.6.1 Audit trail DLLs

Two audit trail DLLs and their source code are supplied with the Blaise system. The first is `audit.dll` and its source code file is `audit.dpr`. This DLL stores all audit trail information for all forms in the database in one file, and is shown below. These files are located in the `\Samples\Dll\Audit` folder of the Blaise system folder.

The second audit trail DLL is auditkey.dll, and its source code file is `auditkey.dpr`. This DLL stores each form's audit trail information in a separate

file, in the format *PrimaryKey.adt*. `Auditkey.dll` and `auditkey.dpr` are in `AuditKey.zip` in the `\Doc\Chapter5\Audit` folder. For more information, refer to the `readme.txt` file that is in the zipped file.

Both DLLs are written in Borland® Delphi™. You can write another audit trail DLL in this or any other system that can produce a DLL. You can use the supplied DLLs as they are, you can modify them, or you can create your own audit trail DLL. Because you can create or modify your own DLL, you have great flexibility in determining the information that you record and the format of that information.

Audit.dll supplied with Blaise

The following is an analysis of `Audit.dll` that is supplied with Blaise:

First there is some initialisation code:

```
library audit;

uses
  SysUtils,
  DepAudit;

var
  AuditFile: Text;
```

The unit *DepAudit.Pas* contains the definition of the interface objects used by the audit trail DLL procedures.

The following function allows the DLL to record various user actions such as the use of an arrow or page key or a mouse click:

```
function CauseToStr(const Cause: Integer): String;
begin
  case Cause of
    AUDIT_PREVFIELD:             Result := 'Previous Field';
    AUDIT_NEXTFIELD:             Result := 'Next Field';
    AUDIT_MOVELEFT :             Result := 'Move Left';
    AUDIT_MOVERIGHT:             Result := 'Move Right';
    AUDIT_MOVEUP:                Result := 'Move Up';
    AUDIT_MOVEDOWN:              Result := 'Move Down';
.
.
.
    AUDIT_SENDFORM:              Result := 'Send form';
    AUDIT_AUTODIAL:              Result := 'Auto dial';
    AUDIT_EXECUTE:               Result := 'Execute';
    else
      Result := 'Unknown-'+IntToStr(Cause);
  end;
end;
```

Here, the status of a field is made available to the audit trail.

```
function FieldStatusToStr(const FieldStatus: Integer): String;
begin
  case FieldStatus of
    1: Result := 'Normal';
    2: Result := 'Don''t Know';
    3: Result := 'Refusal';
  end;
end;
```

The following procedure writes a date and time stamp and the contents of the string constant *s* to the audit trail:

```
procedure WriteToAudit(const s: String);
begin
  try
    writeln(AuditFile, '"'+FormatDateTime('c', Now) + '",', s);
  except;
  end;
end;
```

The following procedure initialises the audit trail session. This is where you can open the audit trail file. The bits of code such as *{$I+}* are compiler directives in Delphi™:

```
procedure AuditTrailInitialization(const AuditInitialization:
          TAuditInitialization); export; stdcall;
var
  AuditName: String;
  ExtensionLen: Integer;
begin
  ExtensionLen:=
          length(ExtractFileExt(AuditInitialization.MetaName));
  AuditName:= ExtractFileName(AuditInitialization.MetaName);
  AssignFile(AuditFile,
          ExtractFilePath(AuditInitialization.DataName)+
          Copy(Auditname,1,length(AuditName)-
               ExtensionLen)+'.adt');
  {$I-}
  Append(AuditFile);
  {$I+}
  if IOResult <> 0 then
  begin
   {$I-}
    Rewrite(AuditFile);
   {$I+}
  end;
  if IOResult = 0 then
  begin
    WriteToAudit('"Start Session"');
  end;
end;
```

The next procedure is used to close the session. This is where you can close the audit trail file.

```
procedure AuditTrailFinalization(const AuditFinalization:
          TAuditFinalization); export; stdcall;
begin
  try
    WriteToAudit('"End Session"');
    CloseFile(AuditFile);
  except
  end;
end;
```

The next procedure records the opening of a form and the key of that form. Because not every form needs to have a primary key, the value of the internal key is also passed to the procedure. When a new form is started, the values of the keys will be empty.

```
procedure AuditTrailEnterForm(const AuditEnterForm:
          TAuditEnterForm); export;stdcall;
var
  s: String;
begin
  s := '"Enter Form:' + IntToStr(AuditEnterForm.InternalKey) +
       '"';
  s := s + ',"Key:' + AuditEnterForm.PrimaryKey+'"';
  WriteToAudit(s);
end;
```

The next procedure records the closing of a form and the key of that form. Because not every form needs to have a primary key, the value of the internal key is also passed to the procedure.

```
procedure AuditTrailLeaveForm(const AuditLeaveForm:
          TAuditLeaveForm); export;stdcall;
var
  s: String;
begin
  s := '"Leave Form:' + IntToStr(AuditLeaveForm.InternalKey) +
       '"';
  s := s + ',"Key:' + AuditLeaveForm.PrimaryKey+'"';
  WriteToAudit(s);
end;
```

The following procedure records a reason for leaving a field, the value and the status of the field that was left, and the name of the field that was left:

```
procedure AuditTrailLeaveField(const AuditLeaveField:
          TAuditLeaveField); export; stdcall;
var
  s: String;
begin
  s := '"Leave Field:' + AuditLeaveField.FieldName + '"';
  s := s + ',"Cause:' + CauseToStr(AuditLeaveField.Cause) +
       '"';
  s := s + ',"Status:' +
       FieldStatusToStr(AuditLeaveField.FieldStatus) + '"';
  s := s + ',"Value:' + AuditLeaveField.FieldValue+'"';
  WriteToAudit(s);
end;
```

The following procedure records the name of the field that is entered and the value of the field when the field is entered:

```
procedure AuditTrailEnterField(const AuditEnterField:
          TAuditEnterField); export; stdcall;
var
  s: String;
begin
  s := '"Enter Field:' + AuditEnterField.FieldName + '"';
  s := s + ',"Status:' +
       FieldStatusToStr(AuditEnterField.FieldStatus) + '"';
  s := s + ',"Value:' + AuditEnterField.FieldValue+'"';
  WriteToAudit(s);
end;
```

The following procedure records audit trail actions such as moving to the next field or pressing the Home key. It also records changing a remark.

```
procedure AuditTrailAction(const AuditAction: TAuditAction);
          export; stdcall;
var
  s: String;
begin
  s := '"Action:' + CauseToStr(AuditAction.Action)+'"';
  if AuditAction.FieldName <> '' then
  begin
    s := s + ',"Field:' + AuditAction.FieldName + '"';
  end;
  case AuditAction.Action of
    AUDIT_REMARKCHANGED: s := s + ',"Remark:' +
    AuditAction.Value + '"';
    AUDIT_SETLANGUAGE:           s := s + ',"Language:' +
AuditAction.Value + '"';
    AUDIT_EDITTYPE:              s := s + ',"Value:' + AuditAction.Value
+ '"';
    AUDIT_OWNMENUENTRY_DLL:      s := s + ',"DLL-info:' +
AuditAction.Value + '"';
    AUDIT_OWNMENUENTRY_EXE:      s := s + ',"Exe-name:' +
AuditAction.Value + '"';
    AUDIT_OWNMENUENTRY_PARALLEL: s := s + ',"Parallel:' +
AuditAction.Value + '"';
    AUDIT_EXECUTE:               s := s + ',"Command:' +
AuditAction.Value + '"';
  end;
  WriteToAudit(s);
end;
```

These *exports* statements make the procedures accessible by the DEP, enabling it to write the information to the audit trail. The seven procedures that are shown in the following sample are necessary for the audit trail to work. If you choose to modify or write your own DLL, you should use the same procedure names as below, keeping the same case for all characters.

```
exports
  AuditTrailInitialization index 1,
  AuditTrailFinalization   index 2,
  AuditTrailLeaveField     index 3,
  AuditTrailEnterField     index 4,
  AuditTrailAction         index 5,
  AuditTrailEnterForm      index 6,
  AuditTrailLeaveForm      index 7;

begin
end.
```

For each session of the DEP:

- The procedures *AuditTrailInitialization* and *AuditTrailFinalization* are called once.
- The procedures *AuditTrailEnterForm* and *AuditTrailLeaveForm* are called once for each form used in the DEP.
- The procedure *AuditTrailEnterField* is called when a field receives the focus.
- The procedure *AuditTrailLeaveField* is called when a field that has the focus loses the focus.
- The procedure *AuditTrailAction* is called when the user performs some kind of action (for instance, when he invokes the help).

! Remember that the information above is only an example of an implementation of an audit trail. The example produces a readable trail of what went on during a DEP session. In a production environment, 'readability' is perhaps less an issue. By defining a standard record layout for each situation that has to be written, the audit trail can be made much smaller, thus consuming less disk space.

5.6.2 Invoking the audit trail DLL

Invoke an audit trail for a survey by making two entries in the mode library file. The mode library file is a system file that controls behaviour and display settings in the DEP. The default system file is `modelib.bml`, and you can edit this file or create other mode library files. Editing this file is covered in depth in Chapter 6, and we only explain the audit trail setting in this chapter.

To invoke an audit trail, open the Mode Library Editor (select *Tools ➤ Modelib Editor* from the Control Centre menu) and open a `.bml` file. From the tree view on the left, select the *Style–Options* settings. The *Audit trail* section appears on the right side of the panel.

Figure 5-5: Audit trail setting in mode library file

- To turn the audit trail on, select the *Make audit trail* box.
- In the *Audit DLL* box, specify the audit trail name and its path.

The information for the audit trail can also be set in the DEP configuration file. See Chapter 6 for more information.

Note that the location of the audit trail itself is determined by the contents of the DLL. In the example, this is done in the procedure *AuditTrailInitialization*.

5.6.3 Contents of the audit trail file

The contents and format of the audit trail are determined by the audit trail DLL you write. The summarisation of the audit trail information is another topic altogether. Manipula can be used to sort through and organise the information into a report.

An example of the audit trail file produced with the audit trail DLL supplied with Blaise is shown:

```
5/28/98 5:04:23 PM Start Session
5/28/98 5:04:23 PM Enter Form:0, Key:
5/28/98 5:04:23 PM Enter Field:ID, Status:Normal, Value:
5/28/98 5:04:29 PM Leave Field:ID, Cause:Next Field,
    Status:Normal, Value: 10
5/28/98 5:04:29 PM Enter Field:Person.FirstName, Status:Normal,
    Value:
5/28/98 5:04:35 PM Leave Field:Person.FirstName, Cause:Next Field,
    Status:Normal, Value:Roger
5/28/98 5:04:35 PM Enter Field:Person.SurName, Status:Normal,
    Value:
5/28/98 5:04:39 PM Leave Field:Person.SurName, Cause:Next Field,
    Status:Normal, Value: Linssen
5/28/98 5:04:39 PM Leave Form:3, Key:10
```

Every indented line is a continuation of the line above it. The indentation in this example is for display purposes only. In the real audit trail file, the lines would continue to the right.

You can see where the contents of each line are specified in the audit trail DLL listing above. For example, the date and time stamp comes from the procedure *WriteToAudit*. You know the value of a field when the cursor enters the field and the value of the field when the cursor leaves the field. For example, at 5:04:29 p.m. the cursor entered the field *Person.FirstName,* where it had an empty value. At 5:04:35 p.m. the field *Person.FirstName* was left with the value *Roger*.

The audit trail file is a text file that can be processed by Manipula programs. Two such programs are supplied that correspond to this audit trail example. These are `auditsummary.man` and `repopulate.man`.

- `Auditsummary.man` provides a summary of the contents of the data entered. It keeps only the last value of any field and deletes all other lines.
- `Repopulate.man` generates a form-specific Manipula program called `Repop.man` that, on a form-by-form basis, can rewrite a data record if data have been lost due to a system crash or another cause.

Check the files for further documentation. If you modify the audit trail format, you might have to modify the Manipula programs as well.

5.6.4 Miscellaneous audit trail information

Turning the audit trail on and off

If an audit trail is implemented for the DEP session, you can turn it on and off by using several methods.

You can use two different mode library files, in which case you will end up using two different versions of the data model's `.bdm` file (the mode library settings are contained in the `.bdm` file). This is not very efficient.

Another and probably better way to do this is to use a DEP configuration file to override the audit trail settings of the mode library file. In this case, the mode library file would have the audit trail turned on, and the DEP configuration file would have it turned off. Since the DEP configuration file is external to the instrument, you do not have to re-prepare the instrument in order to turn the audit trail off.

Another way to turn off the audit trail is to remove or rename the audit trail DLL. If the name of the audit trail DLL is incorrect, the audit trail is turned off.

As the audit trail records every movement and all beginning and ending values of cells that are visited, a large amount of data can be collected, even for one

interview. Be sure to test thoroughly to make sure recording audit trail information has no adverse effect on performance or the available disk space.

While audit trails have their uses in methodological research and for tracing problems, many organisations do not use them once an application has been thoroughly tested and evaluated. This is due to the amount of collected data that have to be archived and managed. Also, you have to find someone to analyse all this information.

5.7 Multimedia Language

Blaise supports multimedia through an extension of its language feature. Multimedia capabilities include the use of audio, graphics, and video. You can use these multimedia capabilities in many ways.

- You can implement audio Computer Assisted Self-Interviewing, also called audio-CASI. A respondent listens to questions, usually through headphones, as the text is displayed on the screen. Audio-CASI facilitates self-administered surveys by enabling subjects with limited reading skills to fully understand the questions. This approach is often used for surveys that ask questions considered to be sensitive.
- Graphics can be used to test a respondent's recognition of images, such as traffic signs.
- Video can be used to provoke respondent reaction to various video clips, such as public health messages.

Blaise plays and displays standard Windows® multimedia files--.WAV for audio, .BMP, .ICO, .WMF, or .EMF for image, and .AVI for video. Creation of these files is done outside of Blaise using other tools.

5.7.1 Implementing the multimedia capability

To implement multimedia capabilities you need to:

- Declare a multimedia language in the instrument.
- For each field where multimedia capability is used, use one or more of the key words SOUND, IMAGE, VIDEO, or ERROR. Because they appear between double quotes in a field language, they are not parsed during preparation. The syntax checker will not find any misspellings.

- For each of the key words SOUND, IMAGE, or VIDEO, refer to a sound, graphic, or video file. For the key word ERROR, refer to a sound file.
- If necessary, adjust the display size and position on screen for graphics and video files. You can also use the key words STRETCH, NOSTRETCH, LEFT, TOP, CENTRE, HEIGHT, WIDTH, and REPEAT.
- Indicate the number of the multimedia language in the mode library or DEP configuration file, and set other settings as needed.
- Supply or create the multimedia files.

5.7.2 Declaring the multimedia language

In the data model, declare a multimedia language.

```
LANGUAGES =  ENG "English",
             MML "Multimedia Language"
```

The code *MML* and the description *Multimedia Language* have no special significance except that they help reviewers of the instrument understand that this is the language where multimedia is implemented.

5.7.3 Multimedia key words in the multimedia language

Once the multimedia language is declared, use one or more of the key words SOUND, IMAGE, VIDEO, or ERROR in that language for each field. The following example is for a question on alcohol use:

```
Alcohol "The next question is about alcohol."
        "IMAGE(party.wmf)" : TContinue;
```

The key word IMAGE is a function with a file name as an argument. The image that is the file `party.wmf` will be displayed.

The following sample question will play a sound file:

```
AlcoholAge "How old were you when you had your first drink of alcohol
           other than a few sips?" MML "SOUND(daq035.wav)" : 0..99
```

When the respondent reaches the field *AlcoholAge,* the sound file `daq035.wav` plays at the same time as the question is presented on the screen.

The *MML* acronym in the example above is not needed because the second language in the field declaration is known to the system as the multimedia language. In the preceding example, the question is displayed at the same time it is spoken. In the following example, the *MML* acronym is needed because the displayed language is missing:

```
AlcoholAge MML "SOUND(daq035.wav)" : 0..99
```

The respondent will hear the sound file, and thus the question, but will not see the question displayed.

Use ERROR to play a sound file when a range error occurs:

```
FIELDS
  "How far do you have to travel to work each day?"
   MML "SOUND(HowFar.wav) ERROR(TooFar.wav)" : 0..500.0
```

The respondent will hear the sound file if the answer is outside the range *0..500.0*.

STOPONKEY or NOSTOPONKEY can be used in the multimedia language to control whether to play a sound or video item completely or stop playing as soon as a key is pressed. Use STOPONKEY to stop the playing of sound or video when a key is pressed. Conversely, NOSTOPONKEY forces the sound or video to play to completion before moving the cursor to the next field.

```
AlcoholAge MML "SOUND(daq035.wav, NOSTOPONKEY)" : 0..99
```

Use STRETCH, WIDTH, and HEIGHT to indicate the size of an image or video file. Use NOSTRETCH to override the multimedia stretch setting in a modelib or DEP configuration file. Use LEFT, TOP, and CENTRE to indicate the position of an image or video file on the screen. Use REPEAT to repeat the instructions in the media text.

For example:

```
IMAGE(hello.bmp,STRETCH,LEFT=20,TOP=10,HEIGHT=200,
      WIDTH=400)

VIDEO(speedis.avi,STRETCH(1.5),CENTRE) REPEAT
```

In the preceding code, the image file `hello.bmp` is stretched to 200x400 pixels, and is displayed with an upper-left position of *20,10*. The video file

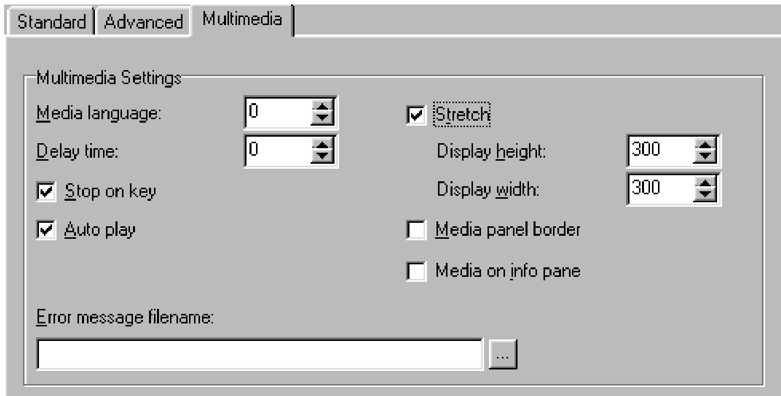
`speedis.avi` is stretched to 1.5 times its original size, is displayed in the centre of the screen, and will be repeated over and over.

5.7.4 Multimedia settings in the mode library file

You need to indicate the number of the multimedia language in the mode library file and specify other multimedia settings as needed. Editing the mode library file is covered thoroughly in Chapter 6, and we explain only the multimedia settings here.

Select *Tools ➢ Modelib Editor* from the Control Centre menu, and the Mode Library Editor opens. Select the *Toggles* settings on the tree view on the left, and then select the behaviour mode to which you want to apply the multimedia settings. Then select the *Multimedia* tab, as shown in the following example.

Figure 5-6: Multimedia settings in the Mode Library Editor



Set the following as needed:

- *Media language.* Specify the number of the language used for multimedia as stated in your data model. For example, if you have three languages listed in the LANGUAGES section and *Multimedia* is the third one listed, the media language would be *3*.
- *Delay time.* This setting is for images only. Specify the number of milliseconds between the presentation of the images.
- *Stop on key.* Select to allow the user to stop a sound file from playing by pressing a keyboard key. If this is not checked, the user can still stop the file using a menu command.

- *Auto play.* Select to have the file begin playing automatically when the user comes to that field.
- *Stretch.* If checked, a picture or video file will take on the values in the *Display height* and *Display width* boxes. If unchecked, the file will display as its default size, regardless of the values in the height and width boxes.
- *Display height* and *Display width.* Specify the height and width, in pixels, of a picture or video file in the DEP window. This applies only if the *Stretch* box is checked.
- *Media panel border.* Select to display a border on the panel on which multimedia files are displayed.
- *Media on InfoPane.* Select to have pictures displayed as part of the InfoPane. If you do not select this option, pictures will appear in a separate window.
- *Error message file name.* Specify the name of a sound file (`.wav`) which will be played when an error occurs.

5.7.5 Other multimedia considerations

Since multimedia is implemented through the language feature, by default it will appear in the language-switching dialog box and the interviewer might switch to it accidentally, and the text multimedia syntax might be displayed.

This can be disabled. *Using Projects ➢ Datamodel properties ➢ Languages* you can specify which of the defined languages will be accessible for the interviewer in the DEP. These languages are accessible in the language dialog or via the previous and next language command in the DEP. By clearing the check box in front of the multimedia language identifier in the lister the text for that language will not be seen on the screen.

Switching multimedia along with spoken languages

Suppose we want to have an instrument in two languages. Consider the following:

```
LANGUAGES =  ENG "English",
             FRA "French",
             MML "Multi Media Language"
```

The display of the question text is taken care of with the English and French language declarations above. Since there is only one multimedia language, the toggling between the two languages for the audio must be done through string fills. For example:

```
LOCALS
  FillSound : STRING

FIELDS
  AlcoholAge "How old were you when you had your first
              drink of alcohol other than a few sips?"
             "Quel âge aviez-vous quand vous avez
               bu votre premier alcohol?"
              MML "^FillSound": 0..99
```

The string local *FillSound* can represent the name of a file. You use the key word ACTIVELANGUAGE to determine which file should be played.

```
RULES
  . . .
  IF ACTIVELANGUAGE = ENG THEN
    FillSound := 'SOUND(alc_ENG.wav)'
  ELSEIF ACTIVELANGUAGE = FRA THEN
    FillSound := 'SOUND(alc_Fra.wav)'

  ENDIF
  AlcoholAge
```

In the example, the file `alc_ENG.wav` plays English speech while the file `alc_FRA.wav` plays French speech.

Audio fills

You can have audio fills. Consider the following:

```
LOCALS
  FillSound1, FillSound2, FillSound3, FillSound4 : STRING

FIELDS
  ChildDrink "How old was ^HeShe when you discovered
               that ^HeShe was drinking?"
             "Quel âge avait-^HeShe quand vous avez
              decouvert qu'^HeShe buvait?"
             "^FillSound1 ^FillSound2 ^FillSound3
              ^FillSound2 ^FillSound4" : 10..20, RF
```

For the multimedia language, there are five parts of the question, each represented by a string local *FillSound1* through *FillSound4,* where *FillSound2* is played twice. To play the audio in the appropriate language, you need the following text computations:

```
RULES
  . . .
  IF ACTIVELANGUAGE = ENG THEN
    FillSound1 := 'SOUND(Howold.wav)'
    FillSound3 := 'SOUND(discover.wav)'
    FillSound4 := 'SOUND(drinking.wav)'
    IF Gender = Male THEN
      HeShe := 'he'
      FillSound2 := 'SOUND(he.wav)'
    ELSEIF Gender = Female THEN
      HeShe := 'she'
      FillSound2 := 'SOUND(she.wav)'
    ENDIF
  ELSEIF ACTIVELANGUAGE = FRA THEN
    FillSound1 := 'SOUND(QuelAge.wav)'
    FillSound3 := 'SOUND(decouvre.wav)'
    FillSound4 := 'SOUND(boire.wav)'
    IF Gender = Male THEN
      HeShe := 'il'
      FillSound2 := 'SOUND(il.wav)'
    ELSEIF Gender = Female THEN
      HeShe := 'elle'
      FillSound2 := 'SOUND(elle.wav)'
    ENDIF
  ENDIF
  ChildDrink            {finally we ask the question}
```

5.8 Question-by-Question Help

In complex surveys you often need to provide the interviewer or respondent with an explanation of terms or phrases used in the question. This is commonly called question-by-question help, or *Q-by-Q* help.

There are two ways to do this in Blaise. The first is to link to the WinHelp utility. The second is to put help text in a help language in Blaise itself.

An advantage of using the WinHelp utility is that the Q-by-Q text can be written outside of the Blaise data model. Thus the Blaise developer does not have to worry about typing or importing this text. A subject matter specialist can type this text in a word processor at the same time the instrument is being developed. Also, the help text does not add to the size of the prepared `.bmi` file.

5.8.1 Using WinHelp

In WinHelp, to link topics to fields in the DEP:

- specify a topic identifier in the fields definition which identifies the topic in the WinHelp file
- set a toggle that enables the link to WinHelp, and
- create WinHelp files.

Settings in Blaise when using WinHelp

There are a few things you have to set in Blaise when you use WinHelp. Because WinHelp is topic based, you have to tell WinHelp which topic to display by specifying a topic identifier.

There are three types of identifiers that can be used to link to topics in the WinHelp file:

- *Help language field text.* The contents of the field text of the help language defined for a field will be used as a topic identifier for the WinHelp system.
- *Field tag.* The contents of the defined field tag will be used for the help.
- *Field name.* The name of the field, without the block path, will be used as the help topic identifier. Because a field always has a name, you must define a topic in the help file for each field.

A topic identifier must start with a letter, but can include letters, digits, spaces, and other characters. If you choose question help in the DEP, the system determines the topic identifier for the current field. These are all set in the mode library file in the Mode Library Editor.

Using the Blaise help language for topic identifiers

Using the Blaise help language for topic identifiers is usually the preferred method of linking to WinHelp. It allows you to reuse topic identifiers in your instrument. If several questions use the same help text, you can specify the same topic identifier for all fields that will use that help text. This cannot be done if you use the field name, because field names must be unique. If the user selects help for a field that has no topic identifier, nothing happens.

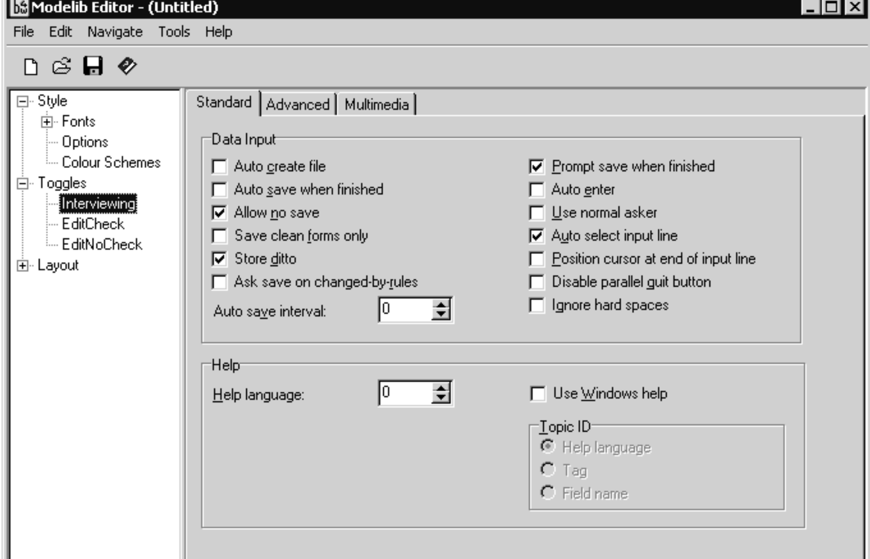
Using Blaise tags as topic IDs also allows reusing topic identifiers in your instrument and the same help text. If you do not use tags for anything else in your instrument, you can specify tags for only those fields for which you want to

display help text. If the user selects help for a field that has no topic identifier, nothing happens. However, if you want to use tags to jump to questions in your instrument, you must make each tag unique. In this case, you cannot reuse help topic identifiers.

With Blaise field names as topic identifiers, and you do not have to create new items (tags or Blaise language) to use as topic identifiers. Also, if you add or change your help text, you do not have to change the data model in any way. One disadvantage of using the field name is that, if you do not include help text for a question and a user selects help for that field, a cryptic error message appears. You either have to specify a help topic for every field, or reuse help topics by using the *Alias* feature of the *Help Workshop* compiler. Another disadvantage is it is not easy to reuse the same help topic for different questions. Because each field name must be unique, you cannot reuse it as a topic identifier. You can solve this by using the *Alias* feature of *Help Workshop* as mentioned above.

To set topic identifiers, open the Mode Library Editor (from the Control Centre, select *Tools ➢ Modelib Editor*), select the *Toggles* settings, choose a behaviour mode, and select the *Standard* tab. The *Help* settings are on the bottom half of the tabsheet.

Figure 5-7: Help settings in the Mode Library Editor



- If your topic ID is *Help language,* specify the number of the help language in the *Help language* box.
- Select the *Use Windows help* box.
- Select a topic identifier in the *Topic ID* box: Help language, tag, or field name.

If the topic identifier is *Help language*, make sure that you have defined a help language in your data model and that the *Help language* box is set to the correct value. In the following example, the help language in the data model is *HLP,* and the value in the *Help language* box of the Mode Library Editor would be *2:*

```
DATAMODEL Age

LANGUAGES = ENG "Plain English",
            HLP "Help topic identifier"

FIELDS
  AgeResp (q1)   "How old is the respondent" "Age" :  0..99
  AgeMother (q2) "How old was the mother when the
                  respondent was born" "Age": 12..50
END
```

In this example, both questions have the same topic identifier in the help language: *Age*. The WinHelp file contains a topic with identifier *Age*. You can leave the help language empty for a field; if you do, nothing will happen when you select question help in the DEP.

If you choose the *Tag* for the topic identifier, the contents of the tag will be used. In this example, each field has a different tag (*q1* and *q2*), so the help file contains a topic with identifier *q1* and a topic with identifier *q2*. You can leave the tag empty for a field; if you do, selecting question help in the DEP does not do anything.

If you choose *Field name* for the topic identifier, the name of the field without the block path will be used as the help topic identifier. Because a field always has a name, you have to define a topic in the help file for each field. In this example: *AgeResp* and *AgeMother*.

Note that when you set toggles in the Mode Library Editor, you can set toggles separately for *Interviewing*, *Data editing (dynamic checking)*, and *Data editing (static checking)*. These three sections correspond to different data entry modes in the DEP. Therefore, you can have different help settings for each of the DEP modes.

5.8.2 Create a WinHelp file

You will need a program to create WinHelp files. Help files are normally created from an RTF file. An example of such a file, `howold.rtf`, is supplied with the Blaise system in the `\Doc\Chapter5\Help` folder.

In our example for the data model `age.bla`, the default name of the help file is `age.hlp`. The system determines the name of the help file by using the name of the `.bmi` file. If you want to give the help file a different name, indicate the help file name in the *Style–Options* settings of the Mode Library Editor. The help file needs to be located in the same directory as the `.bmi` file.

5.8.3 Blaise help language

You can use the language capability of Blaise to write help text. In this method, you declare a help language in the data model, specify the number of the help language in the DEP configuration file, and provide a shortcut or function key in the DEP to invoke the help text. When the help is accessed in the DEP, a window appears that carries the Q-by-Q text. The following `age.bla` data model demonstrates how the help text is specified.

```
DATAMODEL Age

LANGUAGES = ENG "Plain English",
            HLP "Help text"

FIELDS
  AgeResp (q1) "How old is the respondent"
               "Determine the age of the respondent on
                January 1st." : 0..99

  AgeMother (q2) "How old was the mother when the respondent was born"
                 "Determine the age on January 1st
                  of the mother of the respondent.": 12..50

END
```

When you use the Blaise help language facility directly, you can specify a different help language for each DEP behaviour mode. However, the Blaise help language is probably easier to set up for smaller instruments.

5.9 LAYOUT Section

The default Blaise user interface features a page-based presentation. This means that the question text is displayed in the upper part of the screen (in Blaise, this is the *InfoPane*) and a page containing several data entry cells is displayed on the bottom part of the screen (this is the *FormPane)*. The default presentation for data editors does not have an InfoPane and uses the whole screen for the FormPane. Full explanations and examples of the FormPane, InfoPane, and all DEP window components are in Chapter 6.

For many situations the default presentation is fully satisfactory and elegant. Interviewers have found this presentation to be very easy to use because it facilitates navigation and helps increase their understanding of the flow of the instrument.

The Blaise system provides a way to introduce alternative screen presentations. The following samples show different screen presentations that have been used in one instrument. These are from the instrument `layout.bla` found in the `\Doc\Chapter6\Layout` folder of the distribution.

Figure 5-8: Screen layout sample one

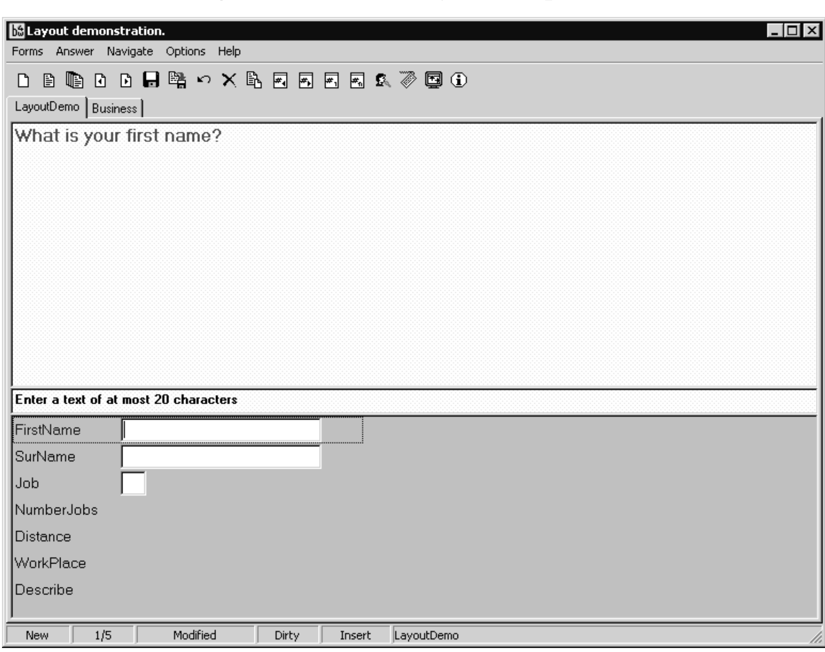


Figure 5-9: Screen layout sample two

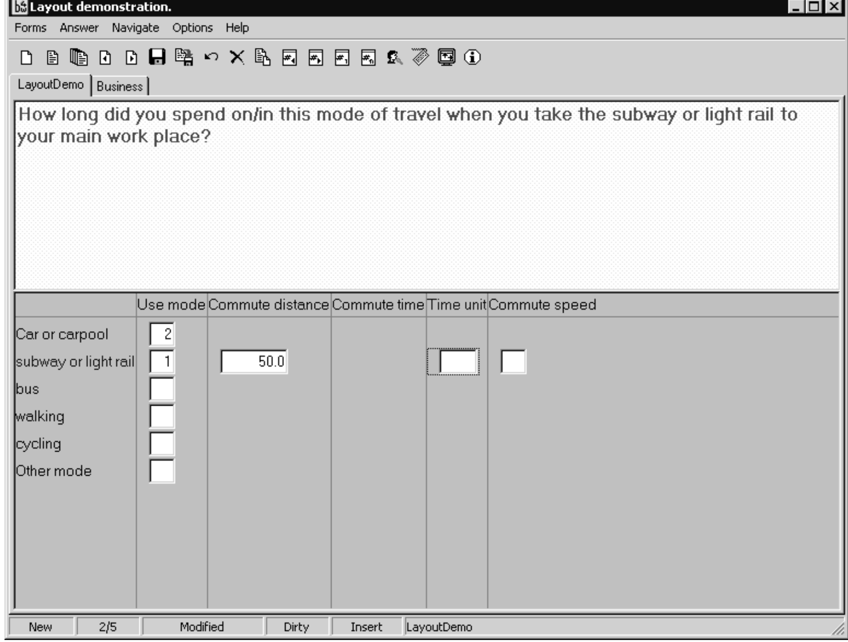


Figure 5-10: Screen layout sample three

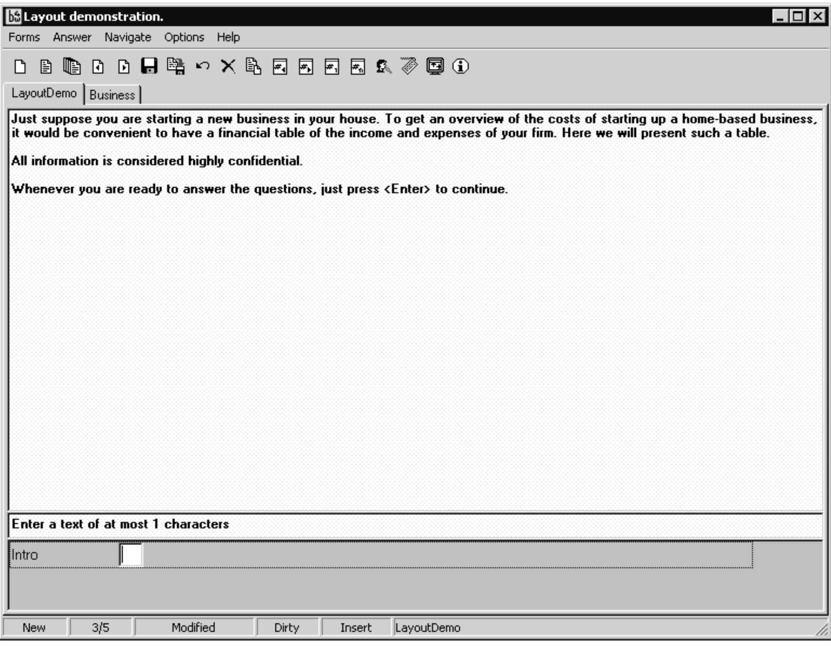


Figure 5-11: Screen layout sample four

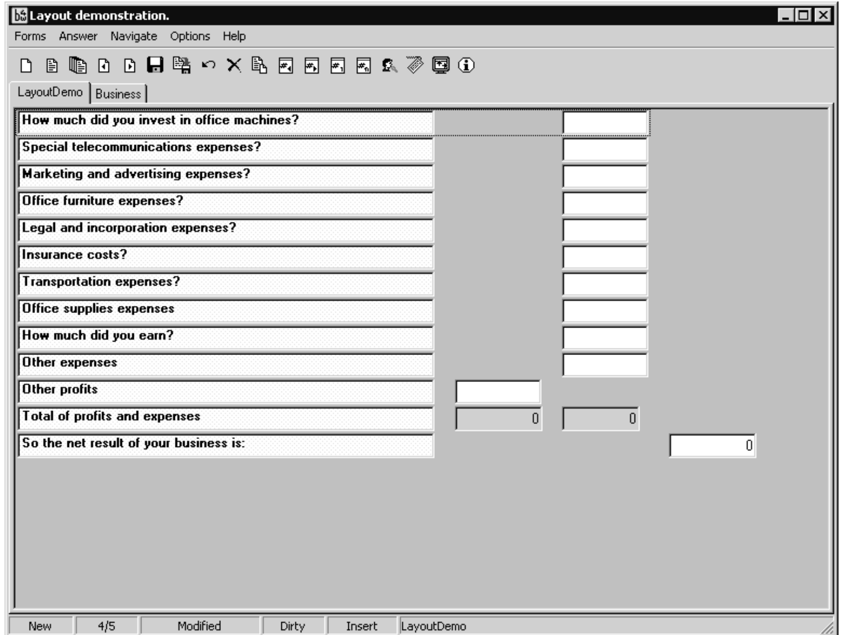
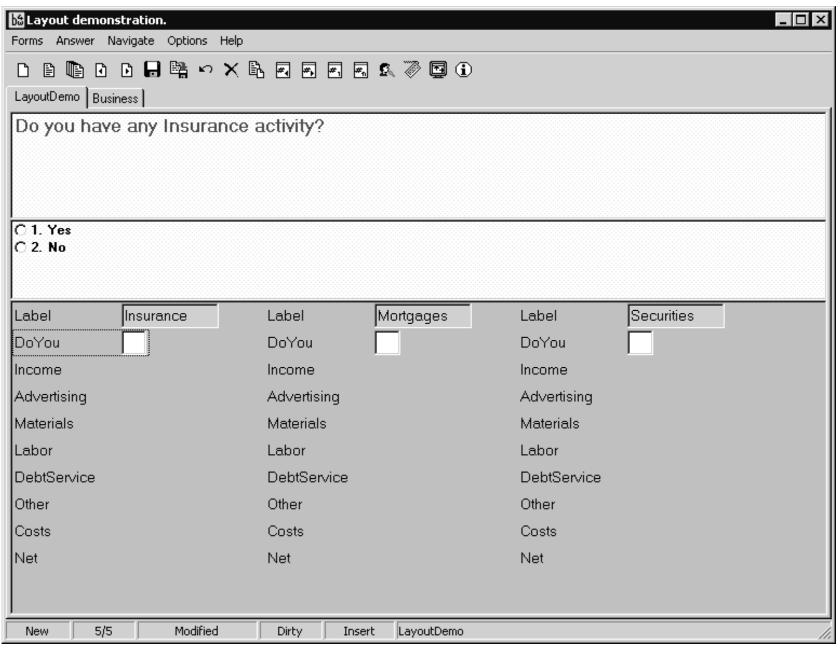


Figure 5-12: Screen layout sample five



Providing alternative screen presentations is a two-step process. The first step is to define a library of possible screen presentations in a mode library file. Creating and maintaining the mode library file is discussed in Chapter 6. Normally an organisation would have one or two individuals develop a mode library file for all the Blaise developers to use.

The second step is to introduce a LAYOUT section in the instrument that the Blaise developers can use to incorporate different screen presentations.

The sample screens shown above came from an instrument called `layout.bla` found in the `\Doc\Chapter6\Layout` folder. The instrument was prepared with the mode library file `modelib.bml` that is also in the `\Doc\Chapter6\Layout` folder. In this mode library file, there are style names for the Grid, the InfoPane, and the FieldPane. A particular screen presentation is achieved by combining the three kinds of specifications. The following table summarises the types of screens and the combination of styles for Grid, InfoPane, and FieldPane that were used.

Figure 5-13: Layout style names for example screens

Type of Screen	Grid	InfoPane	FieldPane
Figure 5-8: Horizontal split screen with 8 rows of fields	Two_x8	For8Rows	For39Char For 80Char
Figure 5-9: Horizontal split screen with 12 rows of fields	_Default TableLayout	For12Rows	_Default TableLayout
Figure 5-10: One question and one entry cell on the screen	Two_x8	DeepPane	For78Char
Figure 5-11: No InfoPane, question text in FormPane	Elev_x20	NoPane	OneLiner1 OneLiner2 OneLiner3 OneLiner4
Figure 5-12: Horizontal split screen with three column FormPane	ThreeCol	For12Rows	For26Char

The style names were chosen to suggest what each style is supposed to do. Style names for *Grid*, *InfoPane*, and *FieldPane* that are present in the mode library file can be used in the layout section. For the example `layout.bla` there are two layout sections. One is at the data model level. The other is at the block level for the block *BMoney*. You can inspect the example file `layout.bla` for details about the layout section for the block *BMoney*. The RULES and LAYOUT sections at the data model level are:

```
RULES {datamodel level}
    Person
    IF Person.Job = Yes THEN
      Commute(WholeName, Person.Distance)
    ENDIF
    Money
    Economic

LAYOUT {datamodel level}
    BEFORE Person
      GRID Two_x8
      INFOPANE For8Rows
      FIELDPANE For39Char
    AT Commute
      GRID Two_x12
      INFOPANE For12Rows
    BEFORE Economic
      NEWPAGE
```

5.9.1 Implementing LAYOUT sections

The LAYOUT section follows the RULES section and is optional. If there is no LAYOUT section, the default styles from the mode library file are used. The

default styles are the first listed styles in the mode library file for *Grid*, *InfoPane*, and *FieldPane*.

If a LAYOUT section is used, all screen layout instructions are taken from the LAYOUT section. Layout elements that are allowed for the RULES section, such as NEWPAGE, NEWLINE, DUMMY, and NEWCOLUMN, are ignored if they are present in the RULES section and a LAYOUT section is also present. A LAYOUT section uses three kinds of key words; location key words, layout style key words, and layout element key words (these are covered in this section).

5.9.2 Location key words

Location key words specify where styles are to take effect. Location key words are summarised below. In the following paragraphs, a field name can be an elementary field name or a block field name:

- AT: The layout instruction applies only to the field. Note that, in addition to field names, the AT instruction can be applied to the key words BLOCKSTART and BLOCKEND.
- BEFORE: The layout instruction applies to any following fields.
- FROM TO: The layout instruction applies from one named field to another named field.
- AFTER: The layout instruction applies after the named field.

Location key words must be followed by layout element key words or layout style key words.

5.9.3 Layout style key words

There are three layout style key words:

- GRID: Applies a Grid style name from the mode library file.
- FIELDPANE: Applies a FieldPane style name from the mode library file.
- INFOPANE: Applies an InfoPane style name from the model library file.

Layout style key words must follow a location key word.

5.9.4 Location and layout key words used together

The following example demonstrates the location and layout style key words working together:

```
BEFORE Person            {block}
  GRID Two_x8
  INFOPANE For8Rows
  FIELDPANE For39Char
```

The field *Person* is a block field name. Its layout instructions apply to all elementary fields in the block. Since the location key word BEFORE is used, all following blocks and fields will have the same styles unless other styles are introduced later.

Before the block *Person*, the Grid *Two_x8* is applied. This Grid applies a page with two columns by eight rows. This allows up to 16 fields on one page for the interviewer or the data editor. Technically speaking, there are 16 FieldPanes in the FormPane. A width of 78 characters for the FormPane is assumed with this Grid definition.

The InfoPane *For8Rows* is designed to go with the Grid *Two_x8*. In other words these Grid and InfoPane style names complement each other. The number of lines allowed for question text in the InfoPane complements the number or rows allowed in the FormPane (page).

There are two columns in this FormPane of 78 characters. We would like each column to hold one FieldPane of 39 characters. A FieldPane definition *For39Char* is defined to place several items in this FieldPane, such as field name, answer cell area, remark indicator location, and response label.

5.10 Example Data Models

The following is a list of example data models and other files found in `\Doc\Chapter5` under the Blaise system folder. These files illustrate the points made in this chapter. You can easily prepare and view them:

Figure 5-14: Example files for Chapter 5

File Name	Description
Audit\AuditKey.zip	Contains an audit trail DLL (auditkey.dll) and its source code (auditkey.dpr) that stores each form's audit trail information in a separate file. Also includes a readme.txt file that contains instructions.
Audit\AtLeave.man	Manipula set-up that condenses the audit trail to only those lines where the DEP is leaving a field. This reduces the amount of redundant text, and prepares the file summary.txt for further processing with Repopulate.man and AuditSummary.man.
Audit\AuditSummary.man	Provides a summary of the contents of the entered data. It keeps only the last value of any field and deletes all other lines.
Audit\Repopulate.man	Creates a form-specific Manipula file named Repop.man. It can be used to repopulate a database from an audit trail.
Classify folder	The files in the Classify folder can be used to produce a classification type from a flat source code file. See the Read.me text file for instructions on preparing and viewing these files.
External\Commut17.bla	Main data model for the external files example.
External\Modelist.bla	External data model to state specifications for the external files example.
External\Modelist.asc	ASCII file of external information for external files example.
External\Modelist.man	Manipula program to convert an ASCII file to a Blaise data file for the external files example.
Help\Age.bla	A small Blaise data model from which you can invoke WinHelp help screens.
Help\HowOld.rtf	Source text file for the WinHelp file.
Help\Age.hlp	Resulting WinHelp file created from the HowOld.rtf file.
Lookup folder	The files in the Lookup folder illustrate a few ways to create lookup files. See the read.me file in the folder for instructions.
MultiMedia\Alcohol.bla	Example of multimedia language.
MultiMedia\Party.wmf, frstdrnk.wav	Image and sound files referenced in the alcohol.bla data model.
Procedure\M_of_20.bla	A data model that demonstrates a procedure which selects m distinct digits out of n <= 20.
Procedure\M_of_20.prc	A Blaise procedure which selects m distinct integers out of n where n <= 20. This can be easily extended to a higher number.

6 Data Entry Program

The Blaise® Data Entry Program (DEP) is a multimode user interface that can be used for Computer Assisted Telephone Interviewing (CATI), Computer Assisted Personal Interviewing (CAPI), and Computer Assisted Self-administered Interviewing (CASI), interactive editing, and data entry from paper forms. The program gives you a tremendous amount of flexibility to change the window layout, menu options, Speedbar, function key assignments, fonts, and colours. This chapter covers how to use the wealth of flexibility available in the DEP and how to adapt it to your needs.

6.1 Overview of Screen Design in Blaise®

To configure the DEP correctly, there are a few points to be made about Blaise's approach to screen design.

Split screen for interviewing

Blaise's default interviewing mode is a splitscreen, page-based presentation with question text on the top part of the screen, and fields and answer information on the bottom part of the screen. Typically, there are several fields displayed at a time. This is in contrast to a question-based presentation, which displays one question at a time to the interviewer.

A typical split screen in a Blaise instrument can display from 10 to 20 questions on one page. This means that all of the responses for an interview, even for a long interview, can be displayed in a manageable number of pages.

This approach is very popular with interviewers. They can see the answers to several previous questions and can verify that data have been entered correctly. They can also use keystrokes to page up and down through the instrument; much in the same way that pages can be turned in a paper questionnaire.

Generated screens

Screens in Blaise are not drawn by programmers, they are generated. Generated screens mean that development is less expensive, maintenance is easier, and it is easier to promote organisational standards. It also allows you to switch modes of operation, including styles of screen display, while running an instrument.

This method is also very robust to changes in your instrument. If questions are added or deleted, you just re-prepare the instrument and the system regenerates the screens. If the screens were drawn by hand, then changes in the design of the instrument could be agonising to implement.

Customise the DEP using external configuration files

Even though screens are generated, Blaise gives you a great deal of power to influence screen display and behaviour. You can customise the DEP using external files. One advantage of specifying behaviour and layout in separate files is that you can change the behaviour and appearance of the DEP without affecting the source code of the data model. Thus you can change the DEP interface for the same data model for different user groups.

Controlling screen layout and behaviour in this way also provides tremendous flexibility, since you can adapt your instrument by changing just a few settings. This frees up the developer's time and standardises the interface for your users.

6.2 DEP Window Components

The DEP window is made up of several components:

- the FormPane (or the page)
- the FieldPane
- the InfoPane
- the Menu bar
- the Speedbar
- the Status bar

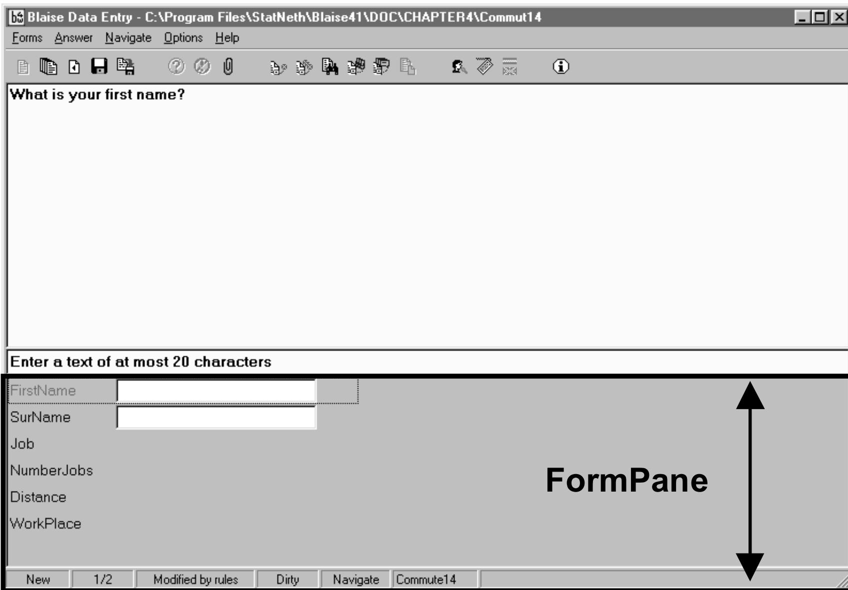
These components are illustrated in the following sections.

There is also an underlying Grid that is used for spacing control for the FormPane and FieldPane. Blaise provides default settings for all of these, but they can be modified by editing a special customisation file called the mode library file (This file is discussed in detail later in this chapter).

6.2.1 FormPane

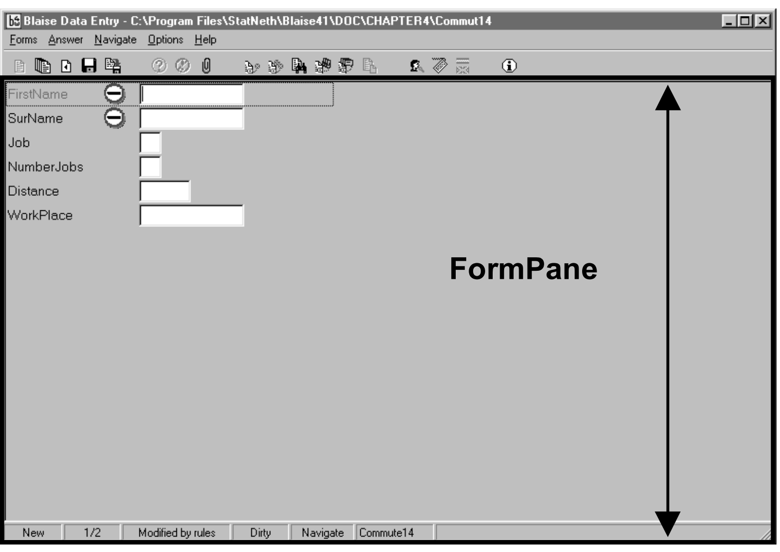
The FormPane, also called the *page,* is the area of the DEP window that displays the fields specified in your data model. This display includes the field names and the spaces provided to enter responses.

Figure 6-1: FormPane



When using interviewing mode, the FormPane will usually occupy the bottom part of the DEP window, though this can be changed. The InfoPane will usually occupy the top part; this is discussed later in this section. For data editing mode, the FormPane usually occupies the entire DEP window, as shown in Figure 6-2.

Figure 6-2: FormPane that occupies the entire DEP window

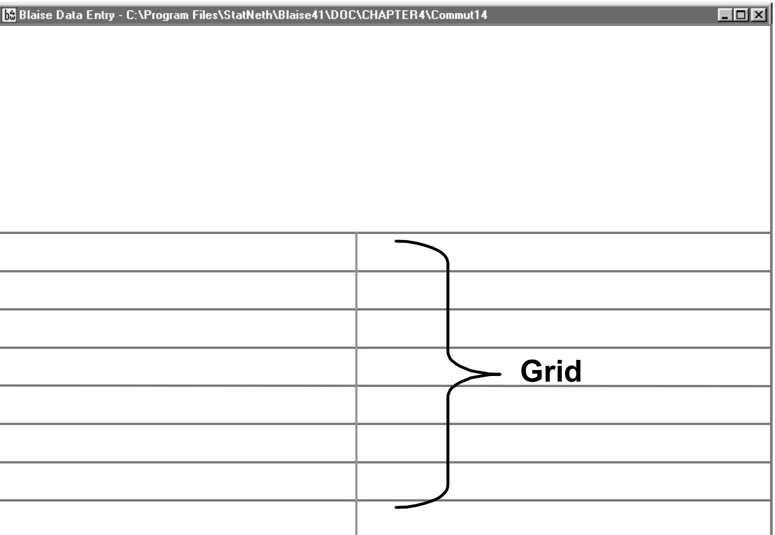


Using the Mode Library Editor, you can adjust the size and layout of the FormPane by adjusting settings for the Grid and the FieldPane. These are described in section *6.5 Mode Library File.*

6.2.2 Grid

The DEP screen is divided by an invisible Grid. The Grid contains cells and the cell size is the base measurement upon which screen elements will be placed on the FormPane.

Figure 6-3: Default Grid for interviewing mode



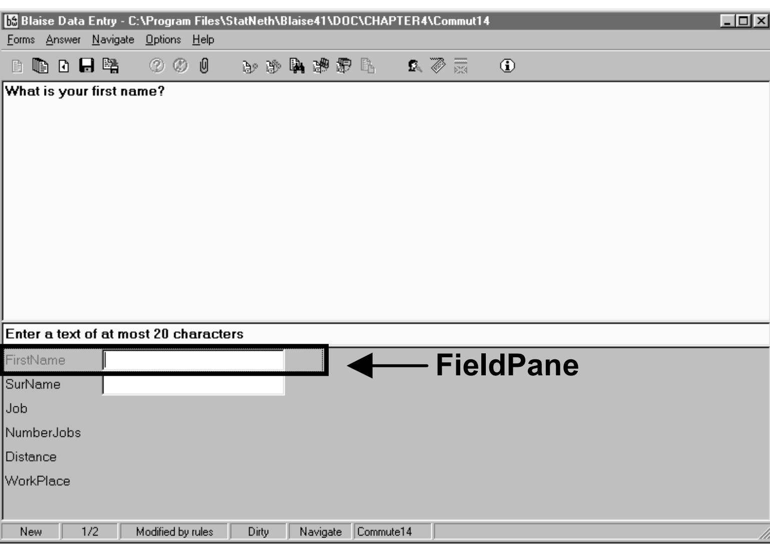
The Grid size is based on pixels. A scaling factor is calculated based on the FormPane font size you use, which you set in the mode library file. The Grid size can extend beyond the size of the DEP window. In that case, the DEP window scrolls when information is entered.

Using the Mode Library Editor, you can adjust the height, width, number of columns, and colour of a Grid. For Grids used for tables, you can also set options for the column and row headings. See section *6.5 Mode Library File* for details.

6.2.3 FieldPane

The FieldPane is an individual unit of the FormPane that contains an individual field. FieldPanes sit in the FormPane area of the window.

Figure 6-4: FieldPane



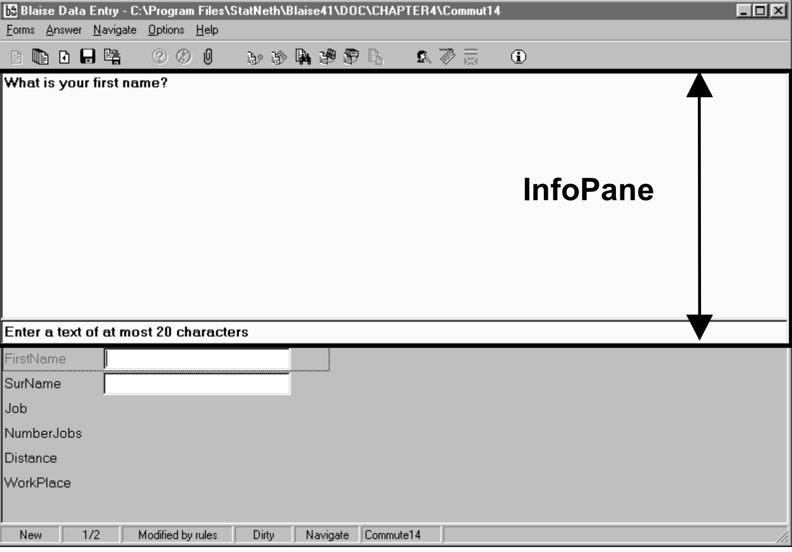
In most cases, each FieldPane will occupy one cell in the Grid, but one FieldPane can occupy two or more cells in the Grid. The FormPane can contain many FieldPanes or just one, depending on your settings.

Using the Mode Library Editor, you can adjust the height and width of the FieldPane. You can also decide which information components appear on the FieldPane, and you can set the position, height, width, and colour of each component. See section *6.5 Mode Library File* for details.

6.2.4 InfoPane

The InfoPane holds the question text and possible answers.

Figure 6-5: InfoPane



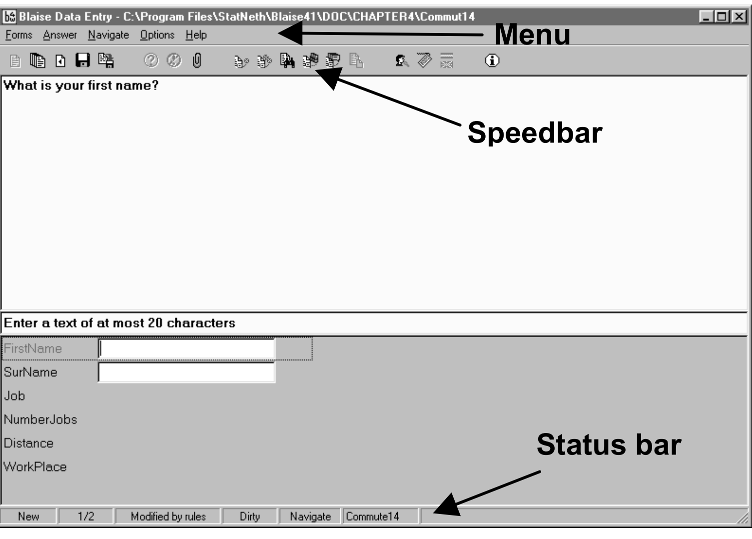
In the default layout, the InfoPane is the top part of the window. You do not have to have an InfoPane in your instrument. Generally, you will not have an InfoPane during data editing mode (as shown in Figure 6-5). The size of the InfoPane is calculated as the FormPane font size multiplied by the InfoPane height.

Using the Mode Library Editor, you can adjust the position and size of the InfoPane, or decide not to display it at all. You can also decide which information components appear on the InfoPane, and you can adjust the position, height, and colour of each component. See section *6.5 Mode Library File* for details.

6.2.5 Menu, Speedbar, and Status bar

The other components of the default DEP window are the Menu bar, the Speedbar, and the Status bar. These items provide options and information for the user when the DEP is running. You can customise the menu and Speedbar choices by editing the DEP menu file, which is described later in this chapter.

Figure 6-6: DEP Menu, Speedbar, and Status bar



Using the Mode Library Editor, you can choose to show or hide the Speedbar and status bar.

6.3 Modes of Behaviour

One of the distinguishing features of Blaise is that an instrument can be run in one of four different modes of behaviour. Because different users have different needs when using the DEP, Blaise allows you to decide and choose the modes of behaviour for each group of users.

You can set behaviours for *routing*, *checking*, and *error reporting*, and you can set behaviour to *dynamic* or *static.* This section describes each of the modes of behaviour.

6.3.1 Routing

Routing determines how the user is led through the instrument. The type of routing determines whether the user is led through the instrument dynamically or has freedom of movement. You can set the DEP to follow either *dynamic* or *static* routing.

Dynamic routing means that the user is led through the instrument by the rules of the application. As answers are recorded, branching and skipping are

automatically executed. This way the user is always on the route as it was programmed by the developer and is not able to go to fields that are off the route. Interviewing almost always uses dynamic routing.

Static routing means that the user has complete freedom to move to any field, whether it is on or off the route. Thus it is inappropriate for interviewers but appropriate for data editors. This might also be useful when entering or editing data that was collected on paper forms, as you might want to go to fields currently off the route.

6.3.2 Checking

Checking determines how rules are checked. *Dynamic checking* means that rules are always checked as data are entered. *Static checking* means that the user decides when to invoke the rules by pressing a function key. Static checking may be useful when editing a very large data model, or when changes are made to an existing form's data.

You have a choice of checking behaviour only if you use static routing. Dynamic routing always enforces dynamic checking.

6.3.3 Error reporting

Error reporting determines if and how errors are displayed to the user.

Dynamic error reporting means that a message is displayed in an error box when an error is encountered. The user must do something immediately to correct or suppress the error.

Static error reporting means that the user does not have to correct errors as they occur. When an error is encountered, an error symbol appears next to the answer cell and an error counter is incremented. The user can then view errors at any time by selecting a menu option or by double-clicking one of the error symbols.

6.3.4 Combining the behaviour modes

The following table summarises the combinations of the behaviour modes. The term *CADI* means Interactive Editing and the term *CAI* means Computer Assisted Interviewing.

Figure 6-7: DEP behaviour combinations

Routing	Checking	Error Reporting	Remarks
Static	Static	Static	Old CADI—Suggest for very large forms.
Static	Static	Dynamic	Impossible.
Static	Dynamic	Static	New CADI—Suggest for most applications.
Static	Dynamic	Dynamic	Impossible.
Dynamic	Static	Static	Impossible.
Dynamic	Static	Dynamic	Impossible.
Dynamic	Dynamic	Static	New CAI—Suggest for training new interviewers or data entry from paper forms.
Dynamic	Dynamic	Dynamic	Old CAI—Suggest for experienced interviewers.

As you can see, there are four possible combinations and four impossible combinations.

The *New CAI* mode, with dynamic routing, dynamic checking, and static error reporting, might be useful for new interviewers who are just learning to use electronic interviewing. You might want them to just learn the survey at first and stay on the route. High-speed data entry might also benefit from this mode. You can enter data following the rules, but concentrate on the errors afterwards.

For almost all interviewers, you would use the *Old CAI* mode with dynamic error reporting turned on. You can then clear up problems when they are encountered.

You define these behaviours in the mode library file, which is discussed in the following section. An example data model that demonstrates the four modes of behaviour is `behaviour.bla`, found in the `\Doc\Chapter6\Behaviour` folder.

It is possible to mix and match screen styles with modes of behaviour. For example, while data editing mode usually does not use the InfoPane, it is possible to have data editing behaviour with a traditional interviewing screen style.

6.4 DEP Customisation Files

There are three Blaise files that you can customise to control the behaviour and appearance of the DEP: a mode library file, a DEP configuration file, and a menu file. The following table summarises the files and what they do:

Figure 6-8: DEP customisation files

	Mode Library File	**Configuration File**	**Menu File**
System default file name	Modelib.bbml	No default file provided	Depmenu.bwm Catimenu.bwm (for CATI instruments)
Purpose	Controls: -Screen layout -Text and colour enhancements -Behaviour settings	Overrides the following settings of the modelib file: -Text and colour enhancements -Behaviour settings	Controls -Available menu choices -Key assignments of menu choices. Allows user-defined menu entries.
How to edit	Edit in Mode Library Editor	Edit in DEP Configuration Program	Edit in Menu Manager
When it must be in place	Before the data model is prepared	Before the data model is run	Before the data model is run

You have a few options for using these files.

- Use the system default file for all your data models.
- Edit and create a new file and use it for all data models.
- Edit and create several different files, and apply different files to different data models.

The settings in these files work in an integrated way. For example, screen layout and font sizes must work together. If you specify a font size that is too large for the screen size, the result could be a DEP window display that is less than optimal. Also, settings in the DEP configuration file override certain settings in the mode library file. The settings must be compatible and the files must be applied correctly to the data model. It is important to understand the relationships between the settings and the files as you customise the DEP window.

6.5 Mode Library File

The mode library file, referred to as *modelib*, is a customisation file that holds settings for:

- Fonts and colours for the DEP
- DEP behaviour options, including multimedia
- Layout of the InfoPane, FormPane, FieldPane, and Grid for the DEP window

Using the settings in the mode library file, you can customise the DEP in many ways. Some examples include:

- Changing and adjusting DEP window colours, either globally for the entire DEP window or for individual parts of the window
- Moving or hiding the InfoPane, even for interviewing mode
- Using descriptive text for table column headings, field names, and other displays
- Adjusting DEP behaviours for interviewing or editing modes
- Creating your own sets of behaviours and layouts

The default mode library file provided with Blaise is `modelib.bml` and can be found in the Blaise system folder.

You can choose either to use the default modelib file settings for layout and behaviour, or you can customise the modelib file to meet your specific requirements. If you use the default modelib file, you need only create, prepare, and run your data model as usual.

If you choose to edit the modelib file, you can define and edit the default settings and apply the defaults to the entire data model, or you can define new layout styles and behaviours, and use those for your data model. You can even define and use different layout styles for different fields of the same data model, all using one modelib file.

Mode Library Editor

The Mode Library Editor is a Blaise program that you use to edit all aspects of the mode library file, including layout and behaviour. To run the Mode Library Editor, select *Tools ➢ Modelib Editor* from the Control Centre menu. It can also be run as a separate program by invoking `Emily.exe`. Details on using the Mode Library Editor are described in the following sections in this chapter.

Modelib file and the .bdm file

Modelib file settings are incorporated in the data model's `.bdm` file. The `.bdm` file is one of two prepared instrument files and is created when the data model is prepared in the Control Centre. The process by which the information from the modelib file gets applied to the data model is as follows:

- Layout and behaviour settings are set and saved to the modelib file, which has a `.bml` extension. This is done in the Mode Library Editor.
- The settings from the modelib file are incorporated in the data model's `.bdm` file. Modelib settings can be incorporated either during the prepare process in the Control Centre, or using the Mode Library Editor.
- When the data model is run by the DEP, the DEP reads the `.bdm` file, and the modelib layout and behaviour settings take effect.
- To use the prepare process in the Control Centre to save the modelib settings to the `.bdm` file, apply the modelib following the criteria outlined in section *6.5.7 Applying a mode library file.*

The following diagram illustrates the modelib process.

Figure 6-9: Modelib process

Control Centre

***Project Options, Store layout*:** - LAYOUT instructions stored in ~mi file	→	`LAYOUT AT FieldName INFOPANE NewPane`
Prepare creates:	→	**bmi file**
- Modelib settings saved to .dm file	→	**bdm file**

Mode Library Editor

Open modelib file library:	→	**bml file**
Open data model: - Reads LAYOUT from ~mi file	→	**bmi file**
View pages	→	**bdm file**
Save data model -Modelib settings saved to ~dm file	→	**bdm file**

Data Entry Program

View: → **bdm file** →

Relationship between the modelib and DEP configuration files

You can set layout *and* behaviour properties in the modelib file. You can, however, use a DEP configuration file to override the modelib behaviour settings. For example, you might want to use one set of behaviours for interviewers, and another set of behaviours for data editors. This makes it very easy to accommodate different users with one prepared instrument.

If you choose to use a DEP configuration file, the settings from the DEP configuration file will override the settings of the modelib file.

You cannot, however, set layout information in the DEP configuration file--all layout information must be set in the modelib file. For more details on the DEP configuration file, see section *6.6 Data model properties*.

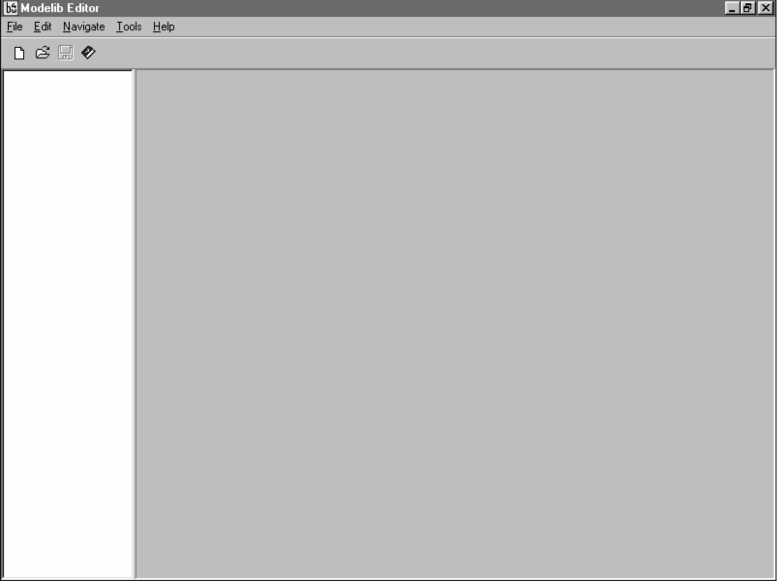
6.5.1 Using the Mode Library Editor

This section describes how to open, save, print, and convert modelib files, and how to open and save data models in the Mode Library Editor.

Open the Mode Library Editor

To open the Mode Library Editor, from the Control Centre select *Tools ➢ Modelib Editor.* The Modelib Editor window opens. If a mode library is identified under Project Options, then the Mode Library Editor will open with this mode library file.

Figure 6-10: Mode Library Editor



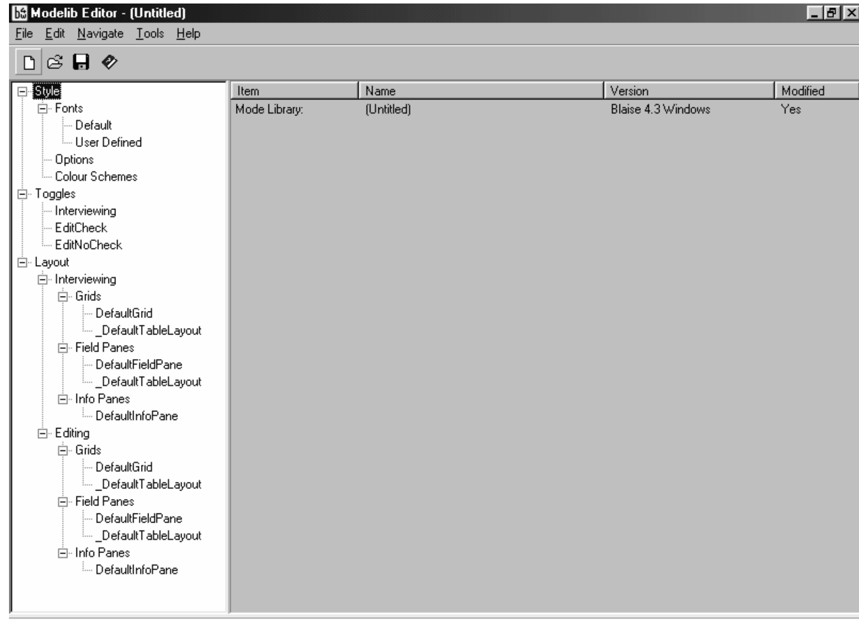
You can use the menus or the speed buttons to access options.

Open a modelib file

To open an existing modelib file, from the menu select *File ➢ Open.* Select a modelib file with a `.bml` extension. To create a new modelib file, select *File ➢ New.*

The branches *Style*, *Toggles,* and *Layout* appear in the tree view on the left. The item, name, version, and modification status of the file appears to the right. Expand the tree view branches by clicking the plus sign next to each one. The following sample shows the window for a new modelib file, with all tree branches expanded:

Figure 6-11: Modelib Editor window for a new modelib file



- The *Style* branch defines fonts, colour schemes, and other DEP options.
- The *Toggles* branch defines behaviours.
- The *Layout* branch defines the size and appearance of Grids, FieldPanes, and InfoPanes.

Settings for each of these branches are described in detail in the following sections of this chapter.

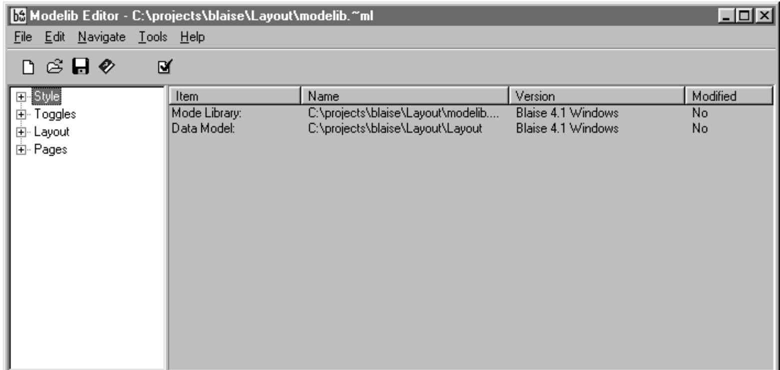
Open a data model in the Modelib Editor

There are two reasons to open a data model in the Modelib Editor. First, you can preview how the layout settings will appear on your instrument's pages. This preview allows you to see how the modelib setttings will look without having to run the DEP.

Second, the modelib settings are saved automatically by the Modelib Editor to guarantee that the Modelib Editor is able to reproduce the DEP pages that conform to the specified layout section in your data model to (the data model's `.bdm` file). You then do not have to re-prepare the data model in the Control Centre to apply the modelib settings—the Modelib Editor will save the modelib information to the `.bdm` file.

To open a data model, select *File ➢ Open Data Model*. Select a data model's prepared `.bmi` file and the name of the data model appears on the right.

Figure 6-12: Data model opened in Mode Library Editor



A new branch called *Pages* appears in the tree view. Use the *Pages* branch to view the pages of your instrument. This is described in detail in section *6.5.5 Viewing pages in the Mode Library Editor.*

The Mode Library Editor prepares the contents of the pages after loading the data model. Note that the mode library file with which you originally prepared your data model is not used. Instead, the currently active mode library file is used. (If you open a data model without opening a modelib file first, the modelib file that the data model was prepared under opens as a new mode library file.)

Save a data model in the Mode Library Editor

To save the modelib information to the `.bdm` file, select *File ➢ Save data model.* If you do not do this, the modelib settings will not be applied to the `.bdm` file, and you will have to re-prepare the data model under the modelib in the Control Centre.

Print modelib settings

You can print your modelib settings by selecting *File ➢ Print* from the menu. This prints a text file that lists all the settings.

Save the modelib file

Once all changes have been made to the modelib file, save it by selecting *File ➢ Save*

Consider saving the modelib file under a new name so that you can always revert to and use the default when necessary. If you rename the modelib file, you must specify to use that modelib according to the criteria listed in section *6.5.7 Applying a mode library file.*

Meta search path option

If you open a data model that uses other data models, you can set an option to have the program search for the meta files of the other data models. Select *Tools ➢ Environment Options*, and enter a search path.

Preview in a separate window option

When checked, the data entry pages will be displayed in a separate window. This can be handy if you want immediate feedback from a change made to one of the options of the mode library. Select *Tools ➢ Environment Options*, and select *Preview in a separate window* check box.

6.5.2 Mode library file: Style settings

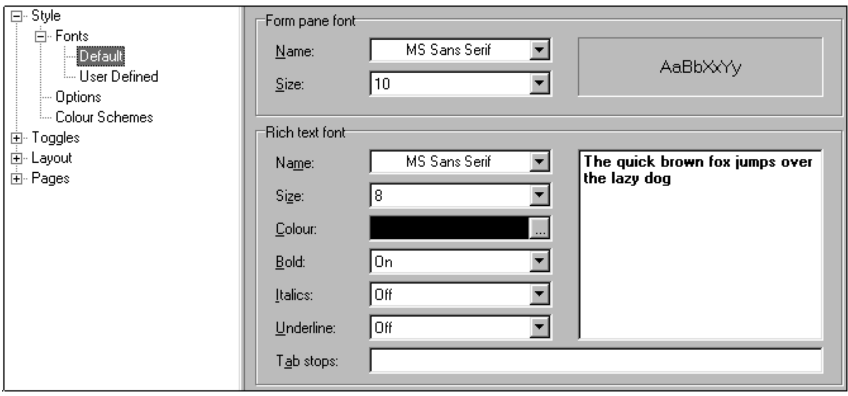
The *Style* branch of the tree view expands to *Fonts, Options,* and *Colour schemes*.

- *Fonts* affect how the text appears in the InfoPane and FormPane. You can change the default fonts and create your own font styles.
- *Options* allows you to set various global options for the DEP.
- *Colour schemes* allows you to select and edit global colour schemes for the DEP.

Style–Fonts–Default fonts

To change the default fonts used in the DEP window, expand the *Fonts* branch and select the *Default* branch, as shown in Figure 6-13.

Figure 6-13: Default font settings



You can adjust the *FormPane font* and the *Rich text font*.

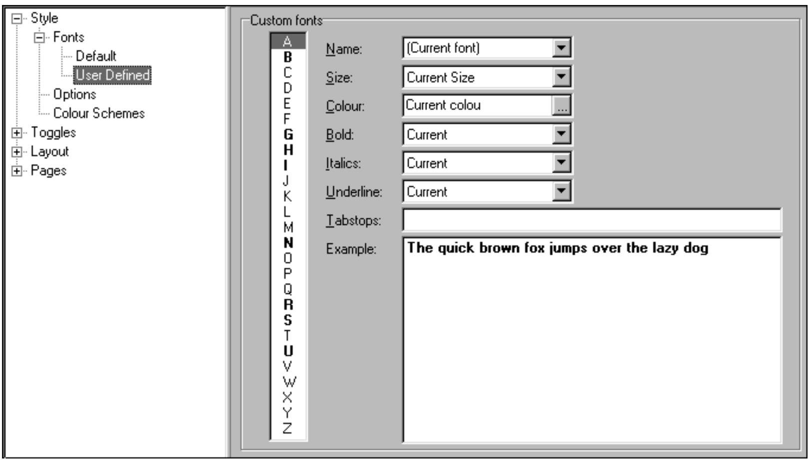
- *FormPane font.* Set the font for the FormPane (the lower part of the DEP window). Select a font name and size. A preview of the settings appears on the right.
- *Rich text font.* Set the font that is used in the InfoPane, and in some elements of the FieldPane. Select a font name and size; select a colour; and turn bold, italics, and underline attributes on or off. In the *Tab stops* box, specify tab stops in pixel increments and separate tab stops with a space. For example, to set tab stops at 25, 75, and 100 pixels, you would type *25 75 100*. The default tab stops are 50 pixels apart. Apply tab stops to your question text by inserting @| in the text. A preview of the settings appears on the right of the screen.

Be careful with the FormPane font size, because it influences the height of the DEP window. The height of the DEP window is the maximum number of lines on the screen (usually 25) times the height of the FormPane font. Increasing the FormPane font size will also increase the size of the InfoPane. If the height of the DEP window is bigger than the computer's screen, scroll bars will appear in the FormPane, but not in the InfoPane.

Style–Fonts–User defined

You can apply your own font attributes to selected text in the InfoPane. To set your own font styles, select the *User Defined* branch under the *Fonts* branch. A list of letters and font attributes appear on the right side of the screen, as shown in Figure 6-14.

Figure 6-14: User defined font settings



Assign attributes to a letter code here, and then enclose data model text within that letter code. When you apply this modelib file to the data model, any data model text enclosed by that code will take on the attributes set here. You can set up to 26 font styles. See Chapter 3 for more information on using the codes in the data model.

Select settings for font name, font size, colour, bold, italics, underline, and tab stop attributes. Specify tab stops in pixel increments and separate tab stops with a space. For example, to set tab stops at 25, 75, and 100 pixels, type *25 75 100*. The default tab stops are 50 pixels apart. Apply tab stops to your question text by inserting @| in the text. There are eight pre-defined letters with the following default attributes. You can, of course, change these default attributes.

Figure Table 6-15: Pre-defined letter attributes

Pre-defined letter	Attribute
@B	Bold
@G	Green
@H	Large font
@I	Italics
@N	Dark blue
@R	Red
@S	Light blue
@U	Underline

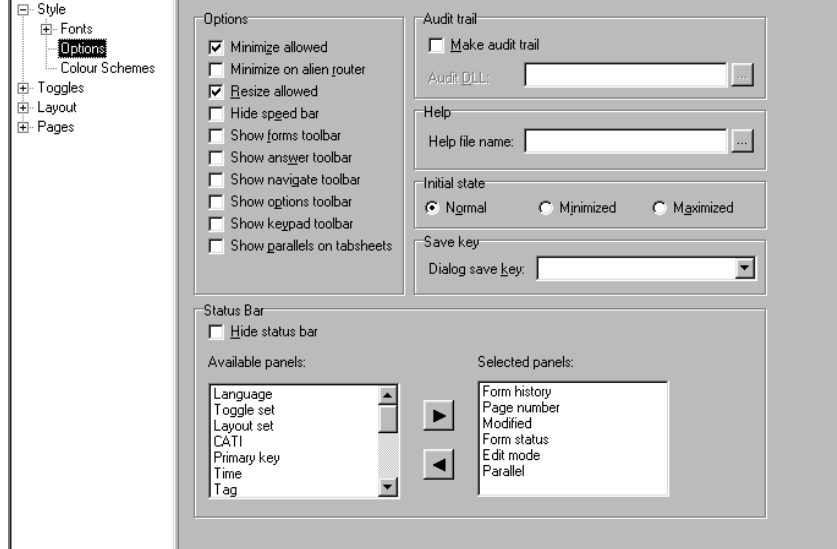
To reset a letter code to its original style on the *User defined* dialog, set its value to *Current*.

Remember that the different attributes will be superimposed upon each other. For example, the code "@BThe @Iquick brown @Ufox jumps @Bover @Ithe @Ulazy dog" translates into **The *quick brown* <u>*fox jumps* over the </u>**lazy dog.

Style–Options

To set various global options for the DEP, select the *Options* branch of the tree.

Figure 6-16: Options settings

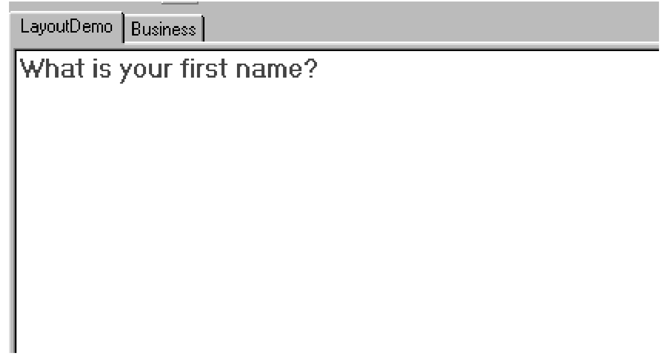


In the *Options* section:

- *Minimise allowed.* Select to allow the user to minimise the DEP window.
- *Minimise on alien router.* Select to minimise the DEP window when an alien router (a user's DLL) is used.
- *Resize allowed.* Select to allow the user to resize or move the DEP window.
- *Hide speed bar.* Select to hide the Speedbar.
- *Forms, Answer. Navigate, Options,* or *Keypad* toolbars. Select to hide or show the following toolbars in the DEP.

- *Show parallels on tabsheets.* Select to display parallel blocks on a tabsheet in the DEP window. Otherwise, the user can access parallel blocks only through the *Navigate ➢ Subforms* menu (or assigned function key) in the DEP. A sample of parallel blocks on a tabsheet is shown below, with a parallel block called *Business.*

Figure 6-17: Parallel blocks on a tabsheet in the DEP



In the *Audit trail* section (see Chapter 5 for more information on the audit trail):

- *Make audit trail.* Select to turn on the audit trail.
- *Audit DLL.* Specify the audit trail DLL name and its path.

In the *Help* section:

- *Help file name:* If you are using WinHelp, specify the help file name here. If a name is not specified, the system will take the name from the `.bmi` file by default. The help file needs to be located in the same folder as the `.bmi` file.

In the *Initial state* section:

- Select whether you want the DEP window to open normally, minimised, or maximised. The *Normal* setting causes the DEP window to open somewhere between maximised and minimised.

In the *Save key* section:

- *Dialog save key*: Select a key that the user can use to close the dialog boxes for remarks and open questions. This provides an alternative to clicking the *Close* or *Save* buttons, or the <Alt+S> or <Alt+C> to close the dialogs.

In the *Status bar* section:

- *Hide status bar:* Select to hide the status bar in the DEP window. If the status bar is not hidden, select the items to appear on the status bar. All available panels are in the *Available panels* list. The panels currently selected to appear are in the *Selected panels* list. Click the arrows to move the panels between the lists. The order in which the panels are listed in the *Selected panels* list is the order in which they will appear in the status bar. Figure 6-18 includes the list of available panels.

Figure 6-18: List of available panels

Form history	When selected the form history is displayed. The form history is new, old, get or dial.
Page number	When selected the number of the current page is displayed and the total number of pages.
Modified	When selected the system displays the modified status of the current form. This status can be modified, modified by rules or read-only.
Form status	When selected the system displays the form status of the current form. This status can be clean, suspect or dirty.
Edit mode	When selected the system displays the edit mode. The edit mode can be navigate, insert or overwrite.
Parallel	When selected the name of the current parallel is displayed.
Language	When selected the identifier of the current language is displayed.
Toggle set	When selected the description of the currently active toggle set is displayed.
Layout set	When selected the description of the currently active layout set is displayed.
CATI	When CATI is active the text CATI is displayed
Primary key	When active the value of the primary key is displayed.
Time	When active the current time is displayed. During CATI interviewing this time is the respondent time (meaning that is corrected to take a possible time difference because of a time zone into account). If there is time difference the time is displayed in a different colour.
Tag	When active the tag of the currently focussed question will be displayed.
User defined 1,.., User defined 9	When selected you can specify a text (with text fills) that will appear on the status bar. When a User defined panel is selected a Status bar tab is displayed under Project|Datamodel Properties in the Blaise Control Centre. On this tab you can enter the value for each of the User defined panels selected.
Field name	When active the field name of the currently focussed field is displayed.
Date	When active the current date is displayed.

Colour schemes

To select a colour scheme for the DEP window, select the *Colour Schemes* branch.

Figure 6-19: Colour scheme settings

A scheme selected here will apply globally to the DEP window. Blaise provides four default colour schemes.

- Select a colour scheme from the *Colour Scheme* section. A preview of the scheme appears in the *Preview* section.
- To change individual items of the colour scheme, select an item from the *Items* list and select a new colour for that item. The *Preview* will reflect the change.
- To apply the scheme to your data model, click the *Apply* button. You must click the *Apply* button after you have made your change.

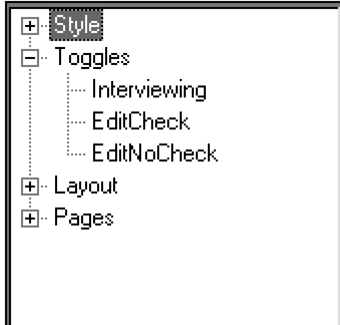
You can also specify colours for specific components of the DEP window by changing the Grid, FieldPane, and InfoPane colours. Colours selected for individual components will override the colour scheme items selected here. When a colour change has been made to an individual item and the *Apply* button is selected, the changes are not made to the colour scheme in the Colour Scheme list box. If you change or just select a colour scheme, any individual item colour changes will be lost. You will need to close the mode library without saving and then reopen the library to bring back your colours.

6.5.3 Mode library file: Toggles

Toggles specify behaviours for the DEP. A *Toggle Set* is a group of behaviour settings named with a *behaviour identifier*. The behaviour identifiers listed in the modelib file are the ones that appear when selecting self-defined data entry modes in the Data Entry Program (from the DEP menu, *Options ➢ Data Entry Mode ➢ Self -defined*).

The default behaviour identifiers are *Interviewing, EditCheck* (dynamic checking), and *EditNoCheck* (static checking), as shown in the following figure:

Figure 6-20: Toggles: default behaviour identifiers



These different kinds of behaviour are used by different kinds of users. For example, *Interviewing* would most often be used by interviewers, and *EditCheck* would most often be used by data editors. It is also common to match different layouts with different behaviour sets.

You can edit the settings for the default behaviour identifiers, add new behaviour identifiers, copy and past new behaviour identifiers, rename, or delete behaviour identifiers.

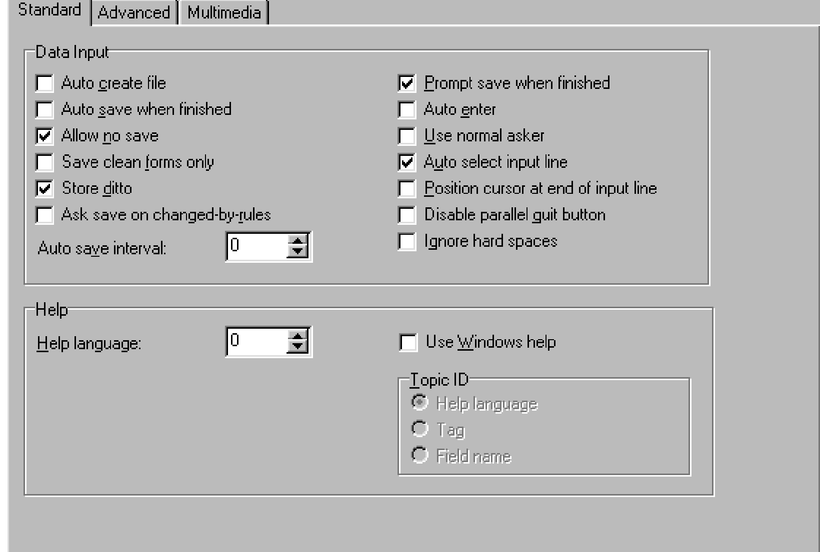
To copy a behaviour definition, select the definition you want from the tree view. With the menu command *Edit ➢ Copy* you make the copy. Now select the *Toggles* node. Use the menu command *Edit ➢ Past* to insert the copy in the tree view. The new behaviour name will have a (2) added to the end of the name. You can change the name by using the *Edit ➢ Rename* command.

There are three tabsheets that contain settings: *Standard, Advanced,* and *Multimedia.* Complete the settings as described in the next section.

Toggles–Standard settings

To set the basic behaviour elements for a toggle set, use the *Standard* tabsheet.

Figure 6-21: Standard tabsheet for Toggles



In the *Data Input* section:

- *Auto create file.* Select to automatically create a Blaise data file, if one does not already exist, when the data model is run.
- *Auto save when finished.* Select to automatically save a form when the end of the form has been reached.
- *Allow no save.* Select to allow the DEP user to exit a form without finishing or saving the form. Be careful with using this option for interviewing.

- *Save clean forms only.* Select to allow only clean forms to be saved. This option can be dangerous if your data model is written in such a way that a clean form can never be produced. Test your data model thoroughly before using this option.
- *Store ditto.* Select to enable the Ditto function, allowing the use of both the Ditto commands of the DEP menu and Ditto assignments in the RULES section. Ditto allows you to copy the values of fields from the previous form to the current one. It is often used for data entry. This option consumes extra memory because the data of the previous forms have to be kept in memory. If this option is not selected, Ditto assignments in the RULES will make the destination fields empty instead of loading the value of the field in the previous form, and Ditto commands will not be available in the DEP menu. This option has no effect and claims no internal memory if you start the DEP with the /X command line parameter (/X causes the DEP to exit automatically after handling one form).
- *Ask save on changed by rules.* Select to enable the save behaviour of the DEP. In this case a form will be saved (or you will be prompted to save it depending on the *prompt save when finished* toggle) if the contents have been changed by the rules only, for instance because of an imputation.
- *Auto save interval.* To minimise data loss during an interview in case of power failure or other such failure, enter in the *Auto save interval* box, the number of minutes between autosaves. Valid numbers are *0* (disabled) to *255*.
- *Prompt save when finished.* Select to have the DEP prompt the user to save when a form is finished. Otherwise, the form is automatically saved on exit.
- *Auto enter.* Select to move the cursor automatically to the next field when the current field has been filled to its maximum width. This is usually used for data entry from paper forms, not interviewing.
- *Use normal asker.* Select to have special routers (CLASSIFY, LOOKUP, and user-defined alien routers) perform only through a short cut or menu command, instead of automatically. These are usually set to perform automatically.
- *Auto select input line.* Select to have the entire contents of the input line selected when the cursor arrives at a field. By default, Windows® selects the contents of the input line as soon as the cursor arrives at the field. If you start typing when the text is selected, the selected text will be overwritten. If you do not want this to occur, uncheck this option.
- *Position cursor at end of input line.* Select this option to position the cursor at the end of the input line when the cursor is on a field. This allows you to easily add text to the end of an answer in the input line.

- *Disable parallel quit button.* Select to remove the quit button from the parallel blocks dialog.
- *Ignore hard spaces:* Select to disable the hard-spaces translation in texts.

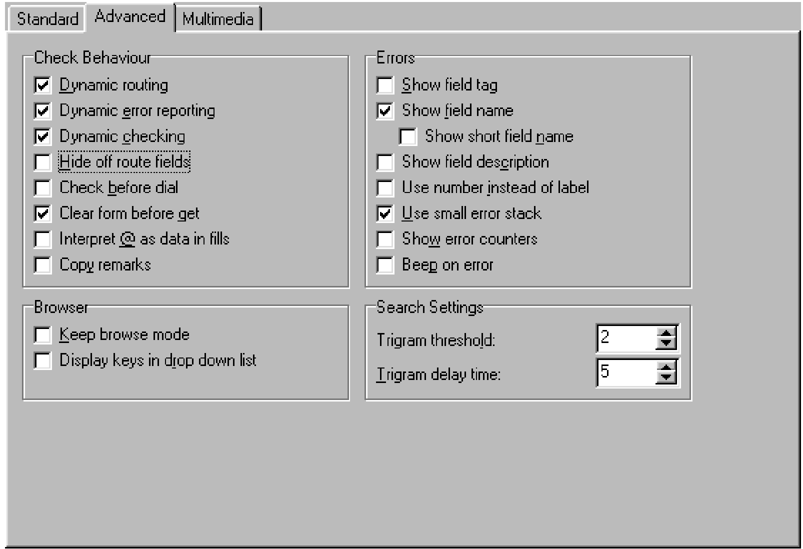
In the *Help* section (see Chapter 5 for more information on *Help*):

- *Help language.* If you have defined a help language within Blaise, or if you are using WinHelp and your topic ID is *Help language,* specify the number of the language used for Help as defined in the data model. (This displays when the DEP menu option *Show Question Text* is selected.) A value of *1* means the first defined language is the help language, *2* the second help language, and so on. A *0* (zero) means no help text and the current language will be displayed in the window.
- *Use Windows® help.* Select to use Windows® Help instead of defining a help language within Blaise.
- *Topic ID.* Select a topic identifier, which tells WinHelp which topic to display for the help. You can choose *Help language, Tag,* or *Field name.* The question help option, <Ctrl-F1>, is only enabled when the *Use Windows help* setting has been enabled and when a *Topic ID* can be determined.

Toggles–Advanced settings

Use the *Advanced* tabsheet to set advanced behaviour elements for a toggle set.

Figure 6-22: Advanced tabsheet for Toggles



In the *Check Behaviour* section:

- *Dynamic routing.* Select to enable dynamic routing.
- *Dynamic error reporting.* Select to enable dynamic error reporting.
- *Dynamic checking.* Select to enable dynamic checking.
- *Hide off route fields.* Select to hide fields that are not on the route.
- *Check before dial.* Select to enable the CATI Call Management System to check the form read from the daybatch before the dial screen appears. This toggle will have no effect if CATI Call Management is not active.
- *Clear form before get.* Select to recheck the data model after the user enters a complete key to get a form. This prevents unwanted results caused by checks performed while the key is still incomplete.
- *Interpret @ as data in fills.* Select this option if you expect the character @ to be part of the data in a fill, such as e-mail addresses. Don’t select this option if you want to use colour codes (@A - @Z) in the fills of your data model, because that would change the way fills are presented on the screen.

- *Copy remarks.* Select this option if you want the data entry program to copy the remarks of a field or block during field and block assignments.

In the *Browser* section:

- *Keep browse mode.* Select to switch the DEP back to browse mode after closing a form that was selected using the Database Browser.
- *Display keys in drop down list.* Select to have key fields appear in a drop down list instead of the default radio buttons, when browsing for forms in the DEP. This is useful if you have defined a lot of keys. When there is not enough room to display the keys as radio buttons, the drop down list is used automatically.

In the *Errors* section:

- *Show field tag.* Select to identify fields in error message boxes by their tags.
- *Show field name.* Select to identify fields in error message boxes by the field name.
- *Show short field name.* Select to identify deeply nested field identifications with a shortened name, using only the first and last block name of the full path name. Other blocks will be noted by a dot. This option cannot be selected unless the *Show field name* option is selected.
- *Show field description.* Select to identify fields in error message boxes by the field description. This can make it easier for the user to select the appropriate field to correct an error. This is also useful for multilingual instruments.
- *Use number instead of label.* Select to identify the values for enumerated and set fields by their code numbers and not by their answer text.
- *Use small error stack.* Select to filter the fields shown in error message boxes. When selected, the DEP shows all the fields involved in a check since the latest display instruction (SHOW, ASK, CLASSIFY, LOOKUP, or user-defined alien router). If this is not selected, error boxes will also involve fields referred to in all conditions leading to the check.
- *Show error counters.* Select to display error counters instead of error icons.
- *Beep on error.* Select to enable the beep feature, which means the computer will beep when an error occurs with dynamic error mode on.

In the *Search Settings* section:

- *Trigram threshold.* Specify the minimum number of trigrams a form or external record must score on before it will be shown within the trigram

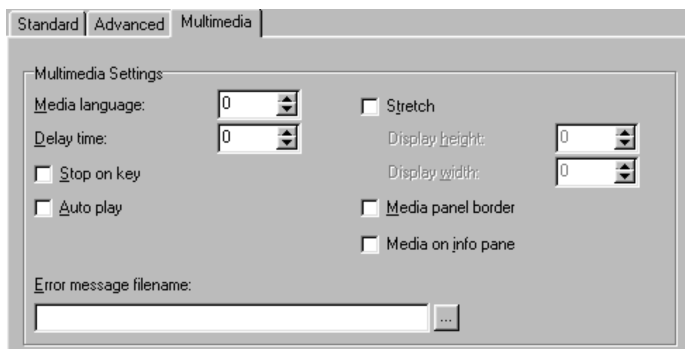
lookup dialog. The lookup dialog is used for browsing forms and looking up external records. A higher number will give faster performance but fewer records as a result of the trigram search. Valid numbers are *0* to *9*.

- *Trigram delay time.* When performing a trigram search, the DEP lets the user enter some text before it starts or restarts searching. The trigram search action is triggered as soon as the user pauses, waits, or hesitates for a certain time interval. In this field, specify the idle time in tenths of a second. Valid numbers are *1* to *20*.

Toggles–Multimedia settings

To set multimedia behaviours, use the *Multimedia* tabsheet.

Figure 6-23: Multimedia tabsheet for Toggles



In the *Multimedia Settings* section (see Chapter 5 for more information on multimedia):

- *Media language.* Specify the number of the language used for multimedia as stated in your data model. For example, if you have three languages listed in the LANGUAGES section and *Multimedia* is the third one listed, the media language would be *3*.
- *Delay time.* This setting is for images only. Specify the number of milliseconds between the presentation of the images.
- *Stop on key.* Select to allow the user to stop a sound file from playing by pressing a keyboard key. If this is not checked, the user can still stop the file using a menu command.
- *Auto play.* Select to have the file begin playing automatically when the user comes to that field.

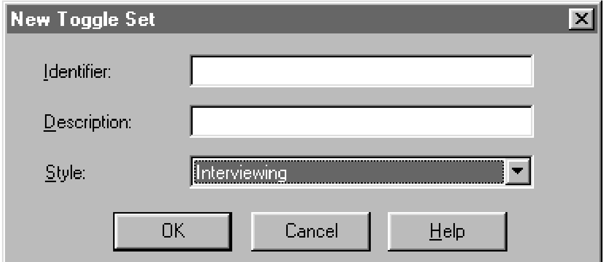
- *Stretch.* If checked, a picture or video file will take on the values in the *Display height* and *Display width* boxes. If unchecked, the file will display as its default size, regardless of the values in the height and width boxes.
- *Display height* and *Display width.* Specify the height and width, in pixels, of a picture or video file in the DEP window. This applies only if the *Stretch* box is checked.
- *Media panel border.* Select to display a border on the panel on which multimedia files are displayed.
- *Media on InfoPane.* Select to have pictures displayed as part of the InfoPane. If you do not select this option, pictures will appear in a separate window.
- *Error message file name.* Specify the name of a sound file (`.wav`) that will be played when an error occurs. If an error message file is not specified with an ERROR instruction in the multimedia language, the file specified here will be the default file that is played.

Add a toggle set

You can add toggle sets to the mode library file.

Select the *Toggles* branch of the tree, and select *Edit ➢ Add Toggle Set* from the menu (You can also right click on the *Toggles* branch.) The *New Toggle Set* dialog box appears.

Figure 6-24: Adding a new toggle set



Complete the items as follows:

- *Identifier.* Specify a name for the toggle set. The identifier must be a unique name containing no spaces, and must start with a letter or an underscore.
- *Description.* Specify a description for the toggle set. This description will appear in the DEP when the user selects a data entry mode.
- *Style.* Select the type of toggle set you want to create. You can select *Interviewing, Data Editing (dynamic checking)*, or *Data Editing (static checking)*. Each style has default settings, which can be changed after you create the set.

Click the *OK* button and the name of the new toggle set appears in the *Toggles* branch.

Apply a toggle set in the DEP

To use a toggle set in the DEP, use one of the following methods.

- Specify */T<Toggle set number>* on the command line of the DEP.
- In the *Data Entry Run Parameters* set in the Control Centre, set the *Toggles* number to the correct toggle set number.
- In the DEP, select *Options ➤ Data Entry Mode ➤ Self Defined*, and select the layout from the *Data Entry Behaviour* box.

Delete a toggle set

To delete a toggle set, click on the appropriate identifier and select *Edit ➤ Delete Toggle Set*. You can also right click on the toggle set to be deleted and select the option from the pop-up menu.

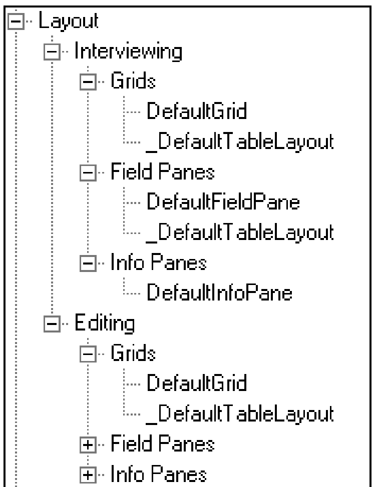
6.5.4 Mode library file: Layout—Grids, FieldPanes, InfoPanes

Layout styles specify attributes of the items on the DEP window. You can apply styles for Grids, InfoPanes, and FieldPanes. For each of these, you can adjust the size, placement, colour, and other characteristics. You can even adjust the properties of individual components of the each item.

A *layout set* is a group of layout settings named with a *layout identifier.* The layout identifiers in the modelib file are the ones that appear when selecting self-defined data entry modes in the Data Entry Program (from the DEP menu, select *Options ➤ Data Entry Mode ➤ Self-defined.*)

There are default layout identifiers for Grids, InfoPanes, and FieldPanes for both Interviewing and editing modes, as shown in the following figure:

Figure 6-25: Layout default identifiers



You can edit the default layouts, and apply those to the data model, or you can define or copy new layouts and then specify those in the LAYOUT section of your data model. You can even choose different styles for different parts of the same data model. If you do not have a LAYOUT section, the data model will use the default settings.

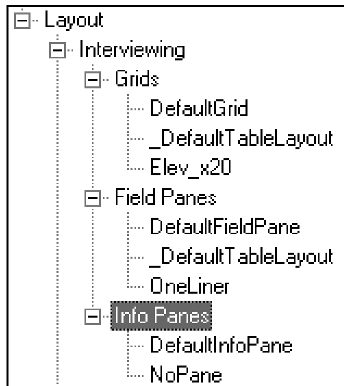
To copy a layout definition, select the definition you want from the tree view. With the menu command *Edit ➢ Copy* you make the copy. Now select the set in which the copy must be pasted. This is at a level one up from where the copy has been made. Use the menu command *Edit ➢ Past* to insert the copy in the tree view. The new layout definition name will have a '(2)' added to the end of the name. You can change the name by using the *Edit ➢ Rename* command.

For example, in the following sample LAYOUT section, specific Grid, InfoPane, and FieldPane styles are applied to several fields:

```
 ...

LAYOUT
  BEFORE Machines
    GRID Elev_x20
    INFOPANE NoPane
  FROM Machines to Earn
    FIELDPANE OneLiner
```

In the modelib file, these would appear on the *Layout* branch of the tree, as shown in the following figure:

Figure 6-26: Layout identifiers in the Mode Library Editor



The layout of the DEP window would then take on the settings for these identifiers. A complete discussion of the data model's LAYOUT section is in Chapter 5.

Add a layout set

You can add a layout set to the mode library file. Select the *Layout* branch of the tree, and select *Edit ➢ Add Layout Set* from the menu. You can also right click on the *Layout* branch.

The *New Layout Set* dialog box appears. This dialog is identical to the *New Toggle Set* dialog box shown in Figure 6-24.

Complete the items as follows:

- *Identifier.* Specify a name for the layout set. The identifier must be a unique name containing no spaces, and must start with a letter or an underscore.
- *Description.* Specify a description for the layout set. This description will appear in the DEP when the user selects a data entry mode.
- *Style.* Select the type of layout set you want to create. You can select *Interviewing* or *Data editing.* Each style has default settings, which can be changed after you have created the set.

Click the *OK* button and the name of the new layout set appears in the *Layout* branch. As with behaviour identifiers, the descriptions you provide for the layout identifiers appear as a choice of *Form and Field Layout* when selecting self-defined data entry modes in the DEP.

Apply layout set in the DEP

To use a layout set in the DEP, use one of the following methods

- Specify */P<Layout set number>* on the command line of the DEP.
- In the *Data Entry Run Parameters* set in the Control Centre, set the *Page layout* number to the correct layout set number.
- In the DEP, select *Options ➤ Data Entry Mode ➤ Self Defined,* and select the layout from the *Form and Field layout* box.

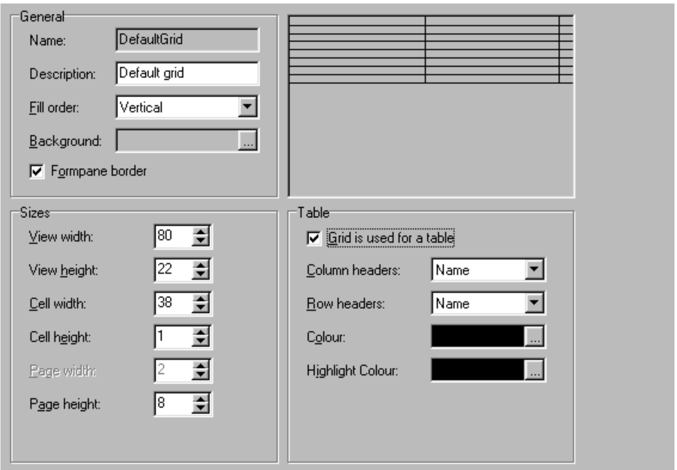
Delete a layout set

To delete a layout set, click on the identifier and select the appropriate option from the *Edit* menu. You can also right click on the identifier to be deleted, and select the option from the pop-up menu.

Layout–Grids

Use the Grids tabsheet to change Grid settings.

Figure 6-27: Grids tabsheet



A Grid is divided into cells. The page builder uses a Grid as a framework on which to place FieldPanes, which hold the questions.

There are a few items to note about the relationship between the Grid and the FieldPane:

- The page builder takes a FieldPane and marks all Grid cells that are occupied by that FieldPane as used. These cells can then no longer be used by another FieldPane.
- Each Grid cell can contain, at most, one FieldPane.
- A FieldPane can occupy more than one cell. You can have a Grid cell width of 40 and a FieldPane width of 80, in which case the FieldPane will occupy two cells.
- A FieldPane will occupy a whole number of cells. For example, suppose the Grid is 80 characters wide, the cell width is 20, and the number of columns (*Page width* setting) is 4. If you have a FieldPane width of 25, one FieldPane will have to occupy two cells, and you will only have two columns in your instrument! So depending on your settings, the number of columns in a Grid (Grid *Page width*) will not necessarily be the same number of columns you see in the instrument.

On the Grids tabsheet, in the *General* section:

- The Grid identifier and its description appear in the *Name* and *Description* boxes. Once the *Name* has been entered it can not be edited.
- *Fill order.* Select *Horizontal* or *vertical* to indicate the order in which the FieldPanes will be placed on the Grid. For the user, this also reflects the direction the cursor will move as answers are recorded.
- *Background.* Select the background colour of the Grid. This is the colour of the FormPane. A colour chosen here will override the Grid colour of the global colour scheme.
- *Formpane border.* When checked the form pane will have a border when displayed on the screen.

In the *Sizes* section:

- *View width.* Specify the number of characters for the width of the DEP window. The default is 80. A preview of this width is in the viewer in the upper right corner of the screen.
- *View height.* Specify the number of lines in your window. The default is 25.
- *Cell width.* Specify the width of the cells of the Grid, in characters.
- *Cell height.* Specify the number of lines in each cell.
- *Page width.* Specify the number of columns in the Grid.
- *Page height.* Specify the total number of lines in the Grid.

If you want to use the Grid as a table, also complete the *Table* section:

- *Grid is used for a table.* Check this box if the Grid will be used for a table definition.
- *Column headers* and *Row headers.* Select what to use for the column and row headers of the table. Select *Name* to use the field name; select *Description* to use the field description, the information that appears in quotation marks after the / in a field definition. When you use the description as the column header in a table, the width of the table column always adapts to current length of the corresponding description. Until now the width of a table column was determined at prepare time and was based on the length of the corresponding field name.

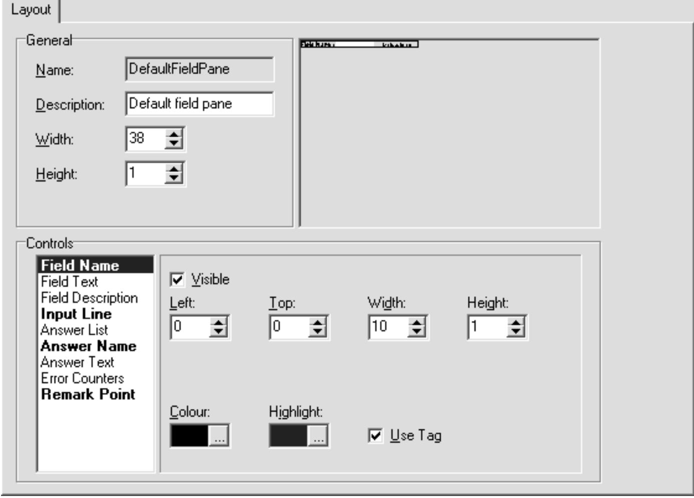
- *Colour.* Select the colour of the font of the column and row headers.
- *Highlight colour.* Select the highlight colour of the active row and active column header.

! If you use your Grid for a table, you cannot use that same Grid for a layout that is not a table. Define different Grids for table and non-table layouts.

Layout–FieldPanes

Use the FieldPanes tabsheet to adjust settings for FieldPanes. FieldPanes hold the questions and question values of the instrument, and are the basic elements with which the page builder fills a page.

Figure 6-28: FieldPanes tabsheet



A preview of the FieldPane appears on the right side of the tabsheet, and changes as you change properties. You can change the zoom of the preview using the *Zoom* box on the speedbar.

In the *General* section:

- The FieldPane identifier and its description appear in the *Name* and *Description* boxes. Only the Description can be edited.
- *Width.* Specify the width of the FieldPane. The width of the FieldPane does not have to correspond exactly to the width of the Grid cell, but a FieldPane will occupy a whole number of cells. Be careful, though, not to make the

FieldPane wider than the width of any Grid on which the FieldPane will be placed.

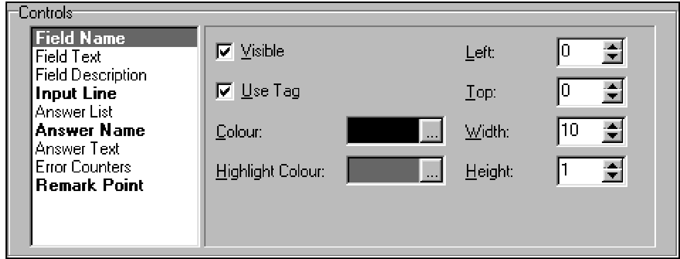
- *Height.* Specify the height of the FieldPane. The height of the FieldPane does not have to correspond to the cell height of the Grid on which it is placed, but a FieldPane will occupy a whole number of cells.

In the *Controls* section, there are nine controls that you can place on the FieldPane, and you can define settings for each control.

To make a control visible on the FieldPane, select the control name in the control list and check the *Visible* box. The properties that you can set for that control then become visible. All visible controls appear bold in the *Controls* lister.

The default FieldPane is set to have *Field Name, Input Line, Answer Name,* and *Remark Point* visible, as shown in the following figure. These defaults can be changed.

Figure 6-29: Default FieldPane controls



The controls and their definitions are as follows:

- *Field Name*. The name of the field as defined in your data model.
- *Field Text*. The question text as defined in your data model. This is a rich text control.
- *Field Description*. The description text as defined in your data model.
- *Input Line*. The space where you type the response in the DEP.
- *Answer List*. A list of possible answers from which you can choose. It is only visible with enumerated and set questions. This is a rich text control.
- *Answer Name*. The identifier corresponding to the answer of an enumerated question. It is only displayed with enumerations.
- *Answer Text*. The text corresponding to the answer of an enumerated question. It is only displayed with enumerations. This is a rich text control.

- *Error Counters*. Error counters indicate which errors a question contains. Error counters can be icons or numbers to indicate route, soft, and hard errors.
- *Remark Point*. The paperclip icon that indicates that a remark has been made for a question.

Define the settings for the FieldPane controls as described in the following list. All possible settings are described, but not all can be applied to each control.

- *Visible.* Specify whether the control is visible or not.
- *Use tag.* For the *Field Name* control, check this box to display the tag that was specified in your data model.
- *Colour.* Select a colour for the control.
- *Highlight.* Select a highlight colour for the control.
- *Left.* Specify the left position of the control, relative from the left of the FieldPane.
- *Top.* Specify the top position of the control, relative from the top of the FieldPane.
- *Width.* Specify the width of the control. You cannot specify a width for the *Remark Point*.
- *Height.* Specify the height of the control. You cannot specify a height for the *Remark Point*.
- *Use for open questions.* For the *Input Line* control, checking this box indicates to the DEP not to open a separate window to type in the answer to an *Open* question. The DEP will use the *Input Line*.

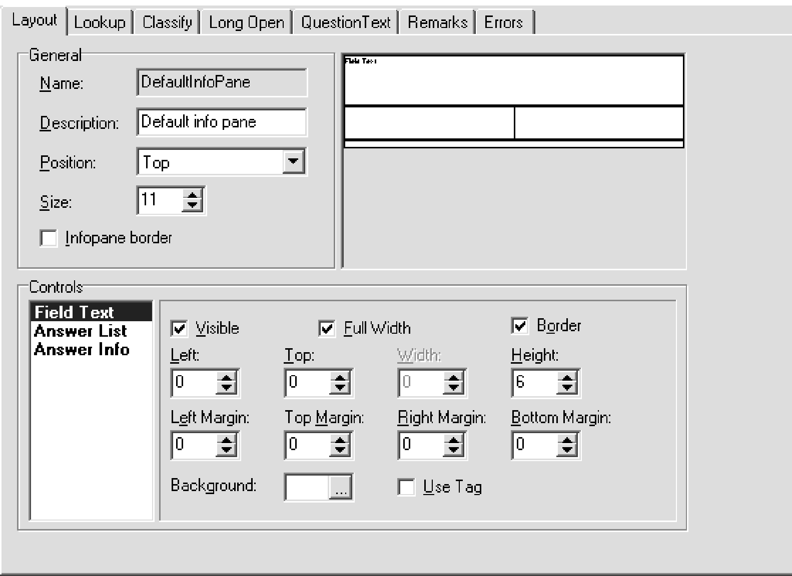
The *Input Line* in the data entry program is now capable to handle multiple lines. It works as follows:

- In the modelib, the height of the *Input Line* control must be set to a value greater than one. The height in the *General* section should be set to the same height as the *Input Line.* The specified height corresponds with the number of lines you want to be visible on the screen.
- You can use the multi-input line also for answering open type questions instead of using the open question dialog. You can indicate this by checking the option *Use for open questions* in the FieldPane.

InfoPanes–Layout tabsheet

To change the way the InfoPane appears on the screen, use the InfoPanes tabsheets.

Figure 6-30: InfoPanes tabsheets



A preview of the InfoPane appears on the right side of the tabsheet, and changes as you change properties. You can change the zoom of the preview using the *Zoom* box on the speedbar.

There are several tabsheets for InfoPanes. The first one, *Layout*, is for the actual InfoPane. The other tabsheets allow you to adjust settings for the various dialog boxes that appear in the DEP.

On the *Layout* tabsheet, in the *General* section:

- The InfoPane identifier and its description appear in the *Name* and *Description* boxes. Only the *Description* can be edited.
- *Position:* Specify the position of the InfoPane on the screen. This can be: *No InfoPane, Top, Bottom, Left* or *Right*.
- *Size:* Specify the size of the InfoPane. Depending on the position you've chosen, this specifies either the width or the height of the InfoPane.
- *Info pane border*: When set the info pane will have a border when displayed on the screen.

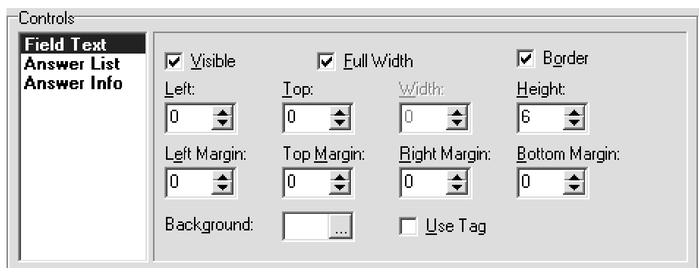
On the *Layout* tabsheet, in the *Controls* section:

There are three controls that you can place on the InfoPane, and you can define settings for each control.

To make a control visible on the InfoPane, click on the control name in the control list and check the *Visible* box. The properties that you can set for that control become visible.

The default InfoPane is set to have all three controls visible, as shown in the following figure. These defaults can be changed.

Figure 6-31: InfoPane controls



The controls and their definitions are as follows:

- *Field Text*. The question text as defined in your data model. This is a rich text control.
- *Answer List*. The list of possible answers from which you can choose. It is only visible with enumerated and set questions. This is a rich text control.
- *Answer Info*. The values being entered in the input line.

Define the settings for the InfoPane controls as described in the following list. All possible settings are described, but not all can be applied to each control.

- *Visible.* Specify whether the control is visible or not.
- *Full Width.* Specify whether the control should use the maximum width possible. When checked, the width property cannot be set.
- *Border.* When set the control will have a border when displayed on the screen.
- *Use Tag.* For the *Field Text* control, check this box to display the tag that was specified in your data model.

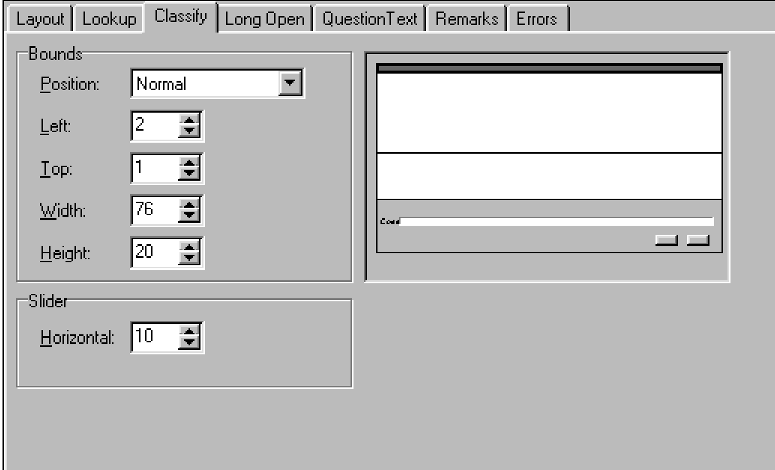
- *Background.* Specify the background colour of the control.
- *Left.* Specify the left position of the control, relative from the left of the InfoPane.
- *Top.* Specify the top position of the control, relative from the top of the InfoPane.
- *Width.* Specify the width of the control.
- *Height.* Specify the height of the control.

In the Answer List control, you can also specify how many columns the answer list should have, whether the code numbers that correspond with the labels should be displayed, and whether the answer list should be balanced (balanced means: the system will try to put an equal number of choices in each column).

In the Answer Info control, you can also specify the left, right, top and bottom margin to be used when displaying the answer info text on the screen. All margins need to be specified in pixels.

On the *Lookup, Classify, Long Open, Question Text, Remarks,* and *Errors* tabsheets, use the remaining tabsheets to adjust the size and appearance of dialog boxes. Each tabsheet is named for a corresponding DEP dialog box. The following sample shows the *Classify* tabsheet, which holds settings for the *Classify* dialog box.

Figure 6-32: Classify tabsheet for InfoPanes



For each dialog, set the following properties.

In the *Bounds* section:

- *Position.* Select a position for the dialog. The position you select here determines which properties (*Left, Top, Width*, and *Height*) you can set. *Normal* will use *Left, Top, Width*, and *Height* as specified. *Centred* will centre the dialog in the DEP window. *Left* will left-align the dialog in the DEP window. *Top* will place the dialog at the top of the window. *Right* will right-align the dialog in the DEP window. *Bottom* will place the dialog at the bottom of the window. *InfoPane* will cause the dialog box to cover the entire InfoPane.
- *Left.* Specify the left position of the dialog box, relative from the DEP window.
- *Top.* Specify the top position of the dialog box, relative from the DEP window.
- *Width.* Specify the width of the dialog box.
- *Height.* Specify the height of the dialog box.

For dialogs that contain *sliders* (the dividing lines within the dialog box), complete the *Sliders* section:

- *Horizontal.* The position of the horizontal slider in the dialog box.
- *Vertical.* The position of the vertical slider in the dialog box. This is found on the *Errors* tabsheet only.

On the *Errors* tabsheet, complete the *Errors* section:

- Select *One Error* or *All Errors* to view how the properties appear for each error viewing mode in the DEP. *One Error* shows the dialog that appears when an error is encountered during data entry. *All Errors* shows the dialog that appears when you select to show errors in the DEP.

Add a Grid, FieldPane, or InfoPane

To add a new Grid, FieldPane, or InfoPane, follow the same procedure for adding a layout set. Select the *Grid, FieldPane,* or *InfoPane* branch of the tree, and select the appropriate option from the *Edit* menu. You can also right click on the branch of the tree.

The identifier you give to a component must exactly match the identifier that is specified in the data model's LAYOUT section.

Delete a Grid, InfoPane, or FieldPane

To delete a Grid, InfoPane, or FieldPane, click on the identifier and select the appropriate option from the *Edit* menu. You can also right click on the identifier to be deleted, and select the option from the pop-up menu.

6.5.5 Viewing pages in the Mode Library Editor

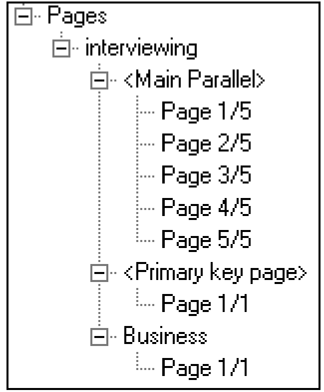
You can preview your pages within the Mode Library Editor as you change the modelib file. This way, you don't have to run the DEP to inspect the results.

Open data model

To view the pages of your data model, first open the data model in the Mode Library Editor by selecting *File ➢ Open data model*. The Mode Library Editor will automatically prepare the contents of the pages after loading the data model.

A new branch called *Pages* appears in the tree. Double click the *Pages* branch to expand it. The tree will expand to show the number of pages in your data model. If you have a primary key or parallel blocks in your data model, these appear as separate branches under *Pages*, as shown in the following figure:

Figure 6-33: Pages branch

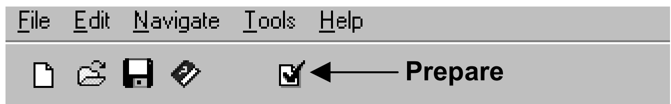


Prepare data model in Mode Library Editor

Each time you make a change to your mode library file, you need to update the layout information so that the Mode Library Editor can build pages for you to view.

Click the *Prepare* speed button on the toolbar, and the page builder updates the layout changes you have made.

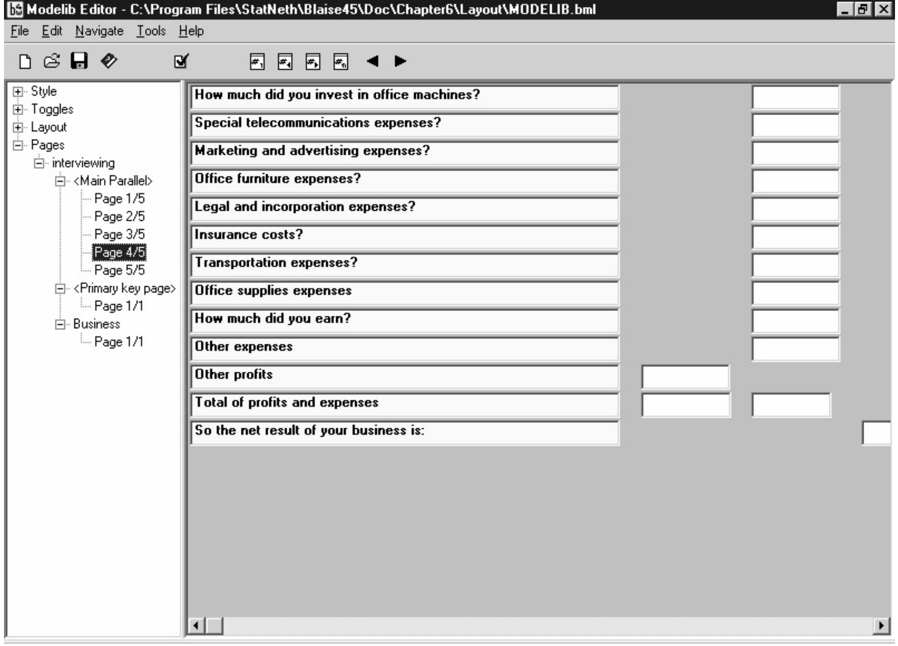
Figure 6-34: Prepare speed button



View pages

Click one of the pages listed in the tree view to see how the page appears in the DEP. A preview of the page appears on the right side of the window, as shown in the following figure:

Figure 6-35: Viewing pages in the Mode Library Editor



You can view each page of the instrument several ways: by clicking the appropriate *Page* branch on the tree, by selecting an option from the *Navigate* menu, or by clicking the appropriate speed button.

In the same way, you can also highlight individual fields to see their layouts. Use the *Navigate* menu, the speed buttons, or click on the field on the preview window.

You might notice there are some settings, such as colour, that you can view without clicking the *Prepare* speed button. These are *run-time* settings. Others are considered *prepare-time* settings and will not be visible until you click *Prepare*.

Page and question properties

You can look at the properties of a page or question of your instrument to see which layouts, behaviour identifiers, and specific Grids, FieldPanes, and InfoPanes were used.

Select a page from the *Pages* branch, and right click on the preview of the page. A pop-up dialog box appears.

To see properties for the entire page, select *Page Properties.* To see properties for the currently displayed field, select *Question Properties.*

A dialog box appears, and shows the individual properties of that page or that field. Each dialog box is displayed in the following figure:

Figure 6-36: Page Properties

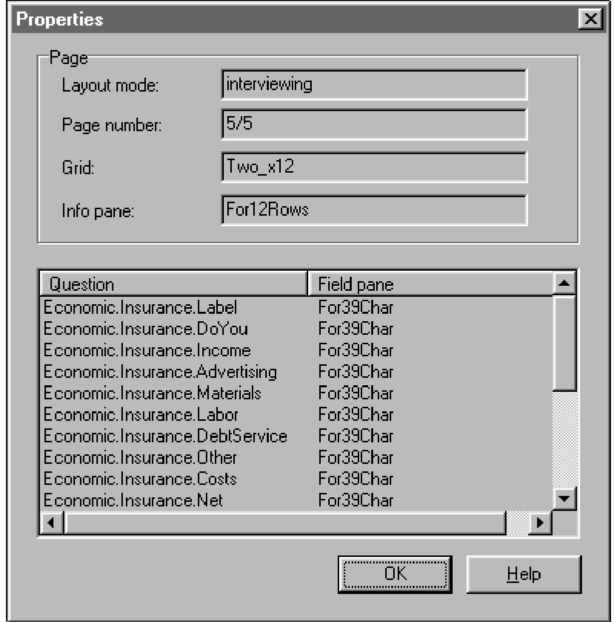
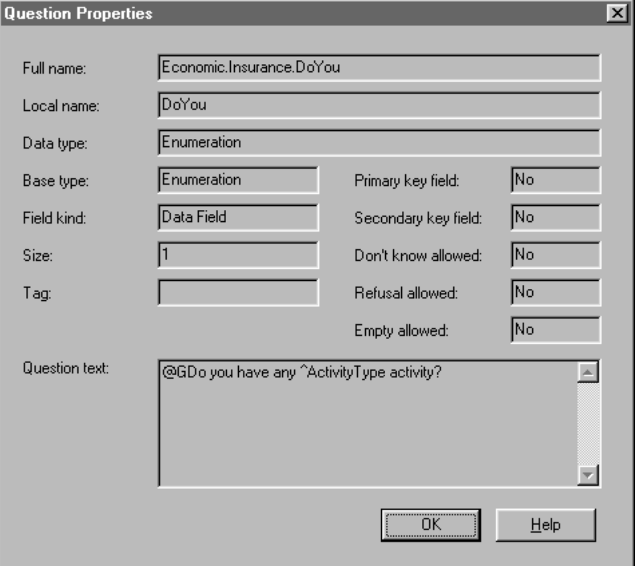


Figure 6-37: Question properties



6.5.6 Common screen layout tasks

This section describes how to accomplish three common tasks that change the appearance of the DEP window:

- Raise or lower the default dividing line
- Change the number of columns in the FormPane
- Remove the InfoPane

The following examples note which modelib settings need to be adapted, and illustrate how the screen elements work together.

Raise or lower the dividing line

A common requirement is to move the default location of the horizontal dividing line of the DEP either up or down. For example, some question texts and interviewer instructions are long. If there is not enough room for the text in the upper part of the screen, the interviewer can still read it, but would have to scroll to see all of it. To avoid this situation, you can move the horizontal line down. Alternatively, if you have short question text you can move the horizontal dividing line up to display more data on the page.

In order to move the line up or down, you have to use a combination of Grid and InfoPane definitions that work together. In `layout.bla`, there is a Grid (*Two_x8)* for 8 rows of fields in the FormPane and an InfoPane (*For8Rows)* that complements the 8 rows. In the same mode library file, there is a Grid definition (*Two_x12*) that allows 12 rows of fields in the FieldPane and an InfoPane (*For12Rows*) that complements the Grid.

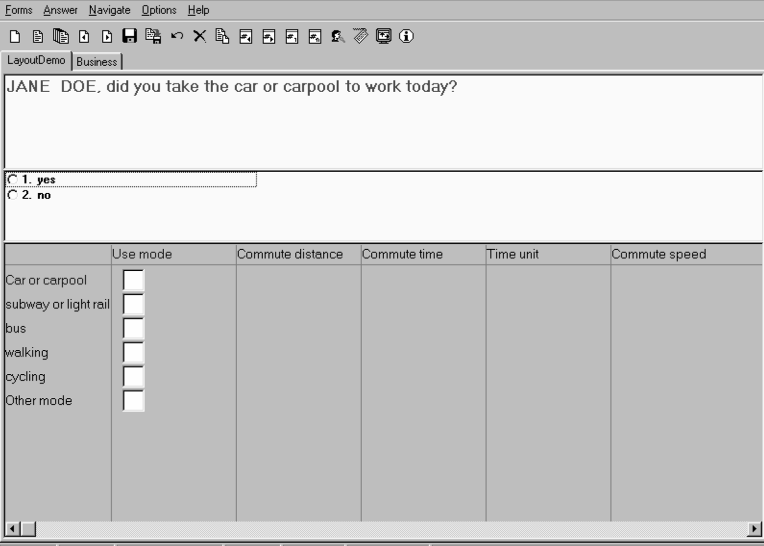
To raise or lower the dividing line, change the size of the InfoPane (the *Size* box in the *General* section of the *InfoPane Layout* tabsheet). You will also usually need to change the height of the Grid (*Page height* setting on the *Grid tabsheet*) to compensate for the height of the InfoPane. If you don't adapt the height of the Grid as well as the InfoPane, you might end up with vertical scrolling bars in the FormPane.

The default InfoPane size is *11*, which divides the DEP window about in half.

- To raise the dividing line, decrease the InfoPane size and increase the height of the Grid.
- To lower the dividing line, increase the InfoPane size and decrease the height of the Grid.

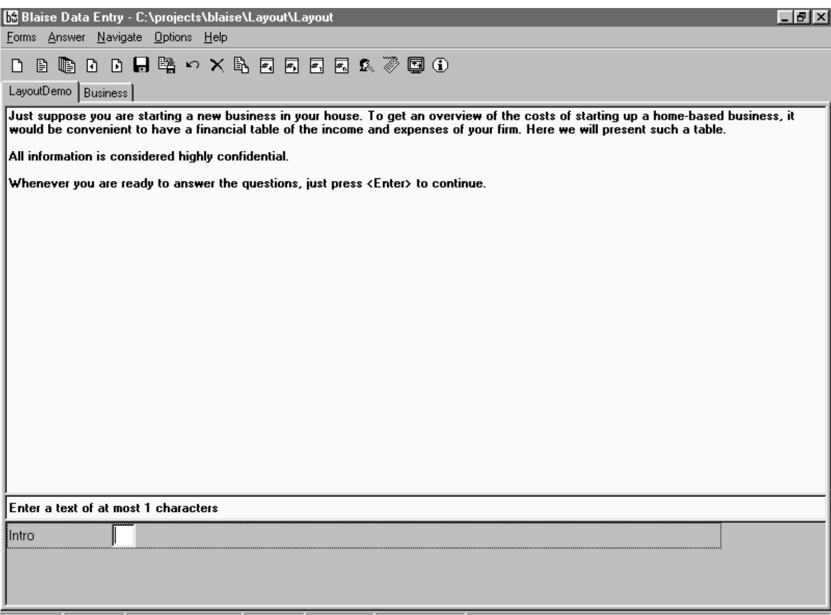
For example, the following sample shows an InfoPane size of *7.*

Figure 6-38: Dividing line moved up



The following sample shows an InfoPane size of *16:*

Figure 6-39: Dividing line moved down

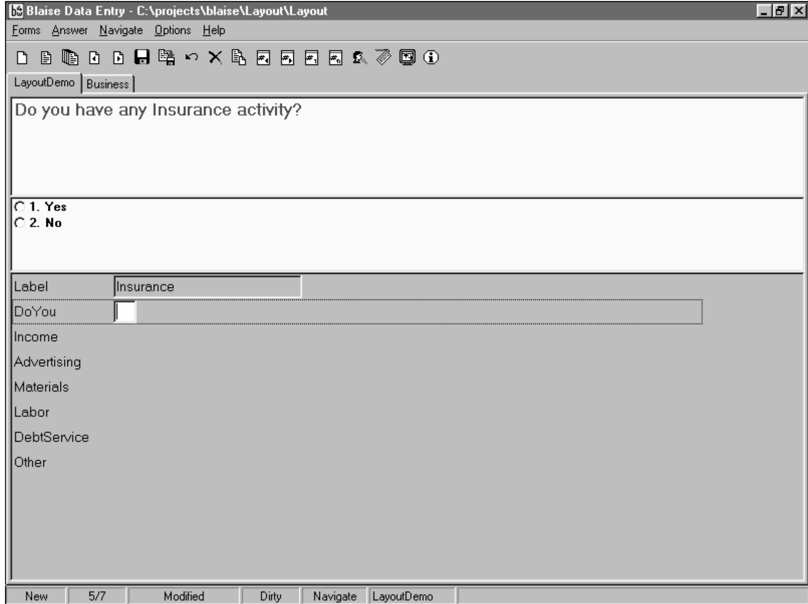


Change the number of columns in the FormPane

To change the number of columns in the FormPane, change the number of columns in the Grid (*Page width* setting.). You might also need to adjust the Grid's cell width, the number of rows in the Grid, the width of the FieldPane used with the Grid, and possibly the FormPane font. Each time you adjust the FieldPane width, also check the properties of the FieldPane's components and adjust them as needed.

For a one-column FormPane, set the Grid's *Page width* to *1*. Again, adjust the Grid's cell width and the FieldPane width to make sure it all fits together properly. The following sample shows a one-column FormPane:

Figure 6-40: One-column FormPane

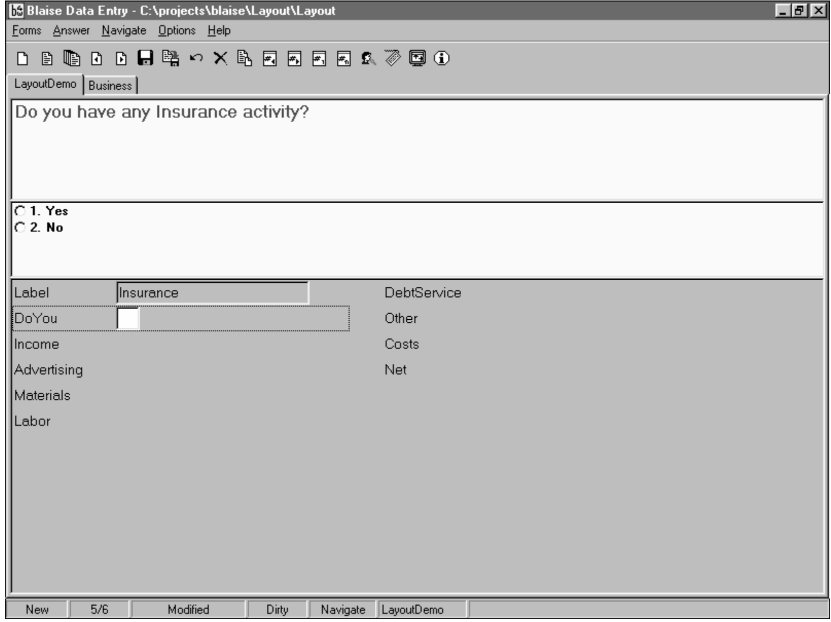


For this example:

- Grid *Page width = 1*
- Grid *Cell width = 78*
- FieldPane *Width = 77*

For a two-column FormPane, adjust the Grid's *Page width* again. The following sample shows a two-column FormPane:

Figure 6-41: Two-column FormPane



For this example:

- Grid *Page width = 2*
- Grid *Cell width = 39*
- FieldPane *Width = 38*

The following sample shows a three-column FormPane:

Figure 6-42: Three-column FormPane

For this example:

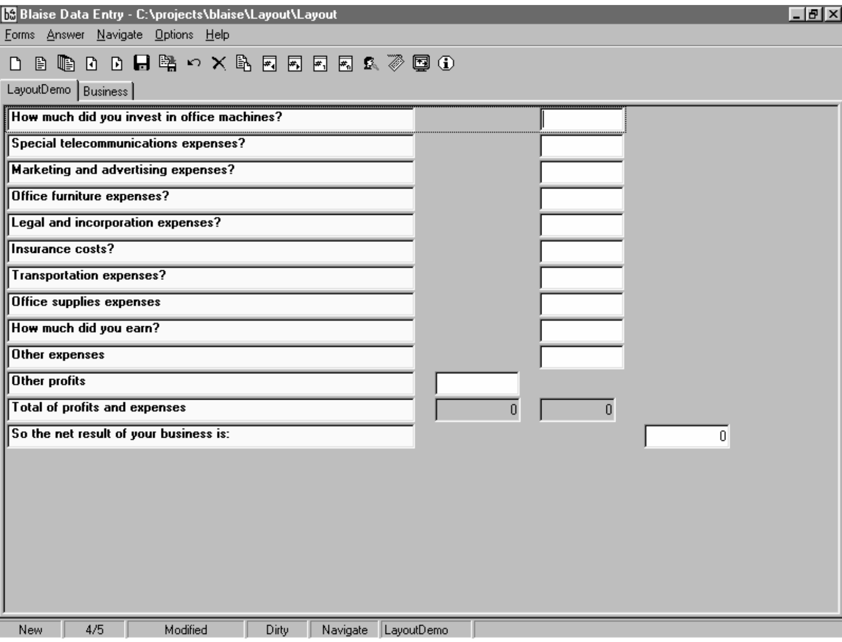
- Grid *Page width = 3*
- Grid *Cell width = 26*
- FieldPane *Width = 25*

Remove the InfoPane

If you do not want the InfoPane to display at all, set the position of the InfoPane to *No InfoPane (*the *Position* setting on the *InfoPane Layout* tabsheet). You might do this if your question text is short, and you want a form-based display.

For example, the following shows a DEP window without an InfoPane:

Figure 6-43: No InfoPane



6.5.7 Applying a mode library file

Applying a modelib file is a two-step process.

Specify which modelib file to use

There are three ways in which Blaise looks for a mode library file and there is an order in which it does so.

- If you specify a modelib file to use in the *Project Options* in the Control Centre, Blaise will use that file. (Specify the file name in the *Mode library* box.)
- If a modelib file is not specified in *Project Options*, Blaise searches the working folder for the *exact* file name `modelib.bml`. Unless you specify a specific file name in *Project Options,* Blaise will only search for and use the file `modelib.bml`.
- If a file is not specified in the *Project Options* and the file `modelib.bml` is not in the working folder, Blaise searches the Blaise system folder for the exact file name `modelib.bml`.

Prepare the data model under the modelib file

Once you have specified the modelib file, prepare your data model. The settings from your modelib file will be incorporated into the `.bdm` file.

6.5.8 Detaching/Attaching a mode library file from a data model

If you have a data model and do not have the modelib file under which it was prepared, you can use the Mode Library Editor to extract the modelib information from the data model.

From the menu select *File ➤ Detach Data Model*. The Data Model line will disappear leaving you with an untitled *Mode Library*. You can now modify the mode library settings and save the file under a different name.

If you have a mode library you wish to apply to a data model, open a mode library and select *File ➤Attach Data Model.* In the Open dialog box select the data model you want. The mode library settings will be applied to the data model when you select *File ➤ Save Data Model.*

6.6 Data model properties

In Chapter 2 on the Blaise Control Centre, various functions related to projects were discussed in Section 2.2.6. One element of the Project system was skipped because it refers to features of the Blaise data model that were only introduced in this chapter. Here we will backtrack to cover the Control Centre's *Project > Datamodel Properties* features.

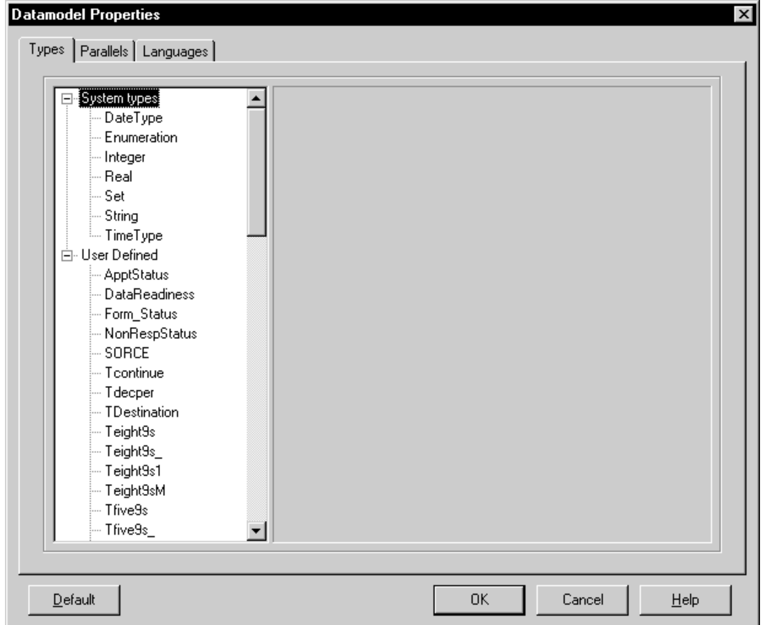
A large number of default display characteristics of a data model can be customised. Examples include:

- Numeric entries displayed with commas or period to separate groups of characters, or as a currency value.
- Date and time values displayed with different separators and with or without AM/PM format.
- Strings displayed with various formatting characters such as a telephone number (999) 999-9999. These formatting patterns are called edit masks.

This is done using settings in the *Data model Properties* form. The properties are then saved to a file <data model name>`.bxi`.

First, open the data model in the Control Centre. Select *Projects ➢ Datamodel Properties* from the Control Centre menu, and the *Data Model Properties* dialog appears.

Figure 6-44: Datamodel properties dialog



6.6.1 Set properties for system and user-defined types of the data model

Select the Types tab. The left part of the dialog displays a tree view, which allows you to select the type definition to view or modify. In the branch *System types* you can select one of the predefined types (date, enumeration, integer, real, set, string, or time type); in the branch *User Defined* you can select one of the user defined types of the data model. The right part of the dialog displays the information for the selected type definition. For each type at least the following information is displayed:

- *Input* type. This displays the base type to be used for the input line in the DEP for the selected type definition. The following base types are available: string, date, time, real, integer, enumeration and currency. Blaise does not support a currency type in the data model, but the input line does support currency for data entry. If the type defined in the data model is a REAL or INTEGER, you

are allowed to switch the base type from real or integer to currency. For instance for a type TTelephone = STRING[20] the base type string will be displayed, and for a type MyNumber = REAL[8,2] the base type real will be displayed. In the latter case you are allowed to change the base type to currency.

- *Alignment.* You can set this to left or right. When set to right, data entry will take place on the right side of the input line. This can be handy when entering numerical data.
- *Dropdown.* You can set this to (none), dropdown, dropdown list, action and date picker (this option is only available for datetype types). When you select dropdown or dropdown list, the input line will have a dropdown list with predefined values that you can select from. Use dropdown if you want a list of values like a list box but you don't want to see the list of values until you push the down arrow button. See the on-line help for more details.
- *Dropdown count.* You can set the number of items to show in the dropdown list. If the actual number in the list is larger, a scrollbar will be used.
- *Password char*. Indicates the character, if any, to display in place of the actual characters in the field.

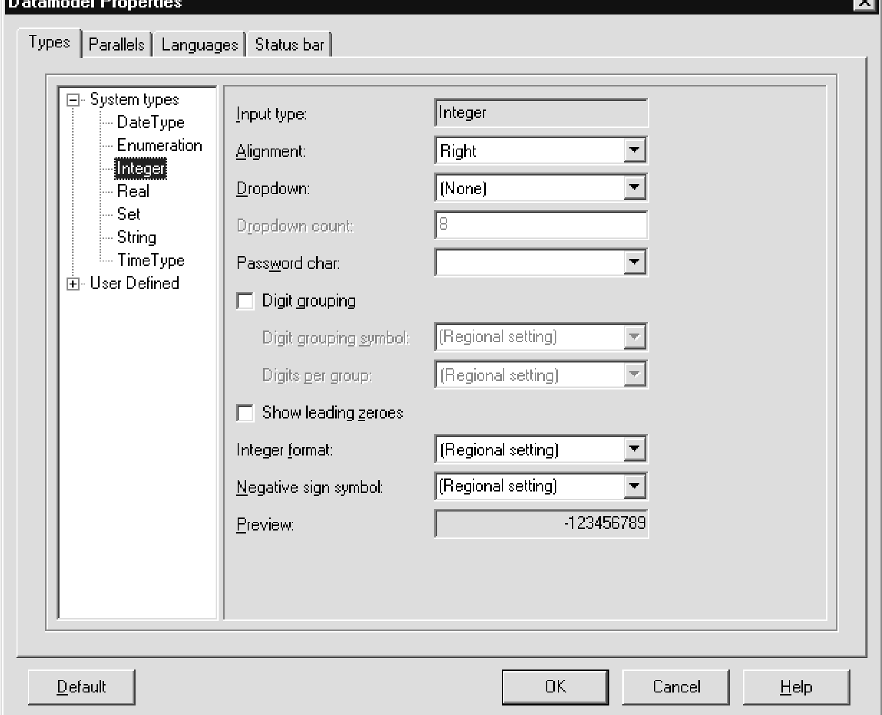
For some of the input types a number of other properties are available.

- *String*. You can set the edit mask and the character to display for a blank character.
- *Time.* You can set the Time separator, time style, hour format, hour leading zero, and AM string and PM string.
- *Real.* You can set the Digit grouping (digit grouping symbol and digits per group), negative sign symbol, real format, decimal symbol.
- *Currency.* You can set the Digit grouping (digit grouping symbol and digits per group), negative sign symbol, decimal symbol (for real based currency fields only), currency symbol, and currency format.
- *Integer.* You can set the Digit grouping (digit grouping symbol and digits per group), negative sign symbol, integer format, and whether to display leading zeros.
- *Date.* You can set the Date separator, date style, leading zero settings, and year style. If the year style is Regional settings or 2-digits, the century button is active to specify the way the system should determine the century in case the user does not specify the century.

- *Enumeration.* You can set the Hide empty categories option. When this option has been set, all categories that have an empty text are not displayed in the answer list in the data entry program.
- *Set.* You can set the separator. You can also set the Hide empty categories option. When this option has been set, all categories that have an empty text are not displayed in the answer list in the data entry program.

In the preview box you can see how the input line in the data entry program will look.

Figure 6-45: Datamodel properties Types page

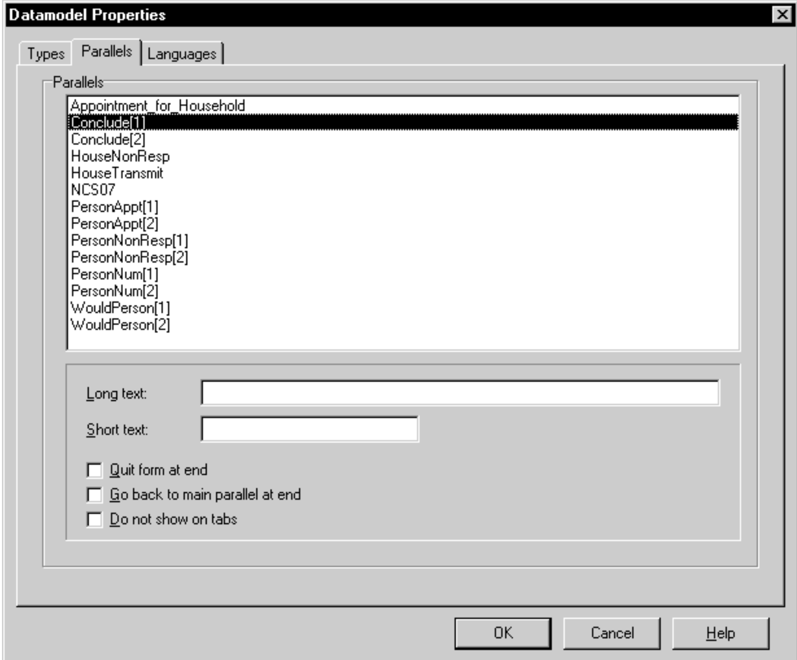


6.6.2 Specify text for parallel blocks

You can specify text that will display in the *Parallel Blocks* dialog of the DEP. By default, when parallel blocks are selected in the DEP, the parallel block name or the identifier that is declared in the data model appears in the dialog box. Here you can specify a more descriptive text for the parallel block.

The names and specified text for the parallels are stored in a file with a `.bxi` extension. Each time you prepare the data model, the system adds or removes parallels and tracks the text that was specified.

Figure 6-46: Data model properties--specifying parallel text



The top part of the dialog displays a list of the available parallels (including the main parallel). If you have specified an array-based parallel, all elements of the array will be displayed in the list. For each entry you can specify a long and a short text to be used in the DEP. This text may contain fills. The fill must be based on a fully qualified field name. A possible subscript in such a name must be a numerical constant.

The long text is the text that has to be displayed in the parallel dialog of the DEP. If no text has been specified, the name of the parallel will be displayed. The short text is the text that has to be used as the caption of the tab for the parallel (Parallels are displayed on tab sheets if the style setting *Show parallels on tab sheets* has been set in the mode library).

You can specify three extra settings:

- *Quit form at end.* When set, the DEP will prompt you to quit the form as soon as the end of this parallel has been reached. You will not be directed to the

parallel forms dialog in this case. When the setting *Disable parallel quit button* has been set in the mode library, you have the same parallel behaviour available during CATI interviewing.

- *Go back to main parallel at end.* When this option is set, the DEP will automatically go back to the active field of the main parallel when the end of the current parallel has been reached.
- *Do not show on tabs.* When this option is set, the parallel will not be displayed on a tab sheet. This setting will be used only when the style setting Show parallels on tab sheets has been set in the mode library.

All mentioned settings are not available for the main parallel. The quit form at end and the go back to main parallel at end options are mutually exclusive.

The system does not check the fills specified in the text—this is done by the DEP. If a field cannot be found in the data model, the fill will be skipped.

To use the parallel text settings in the DEP, make sure that the `.bxi` file is in the same folder as the corresponding prepared data model. If the `.bxi` file is not present or if no text has been specified, the parallel block name will be displayed.

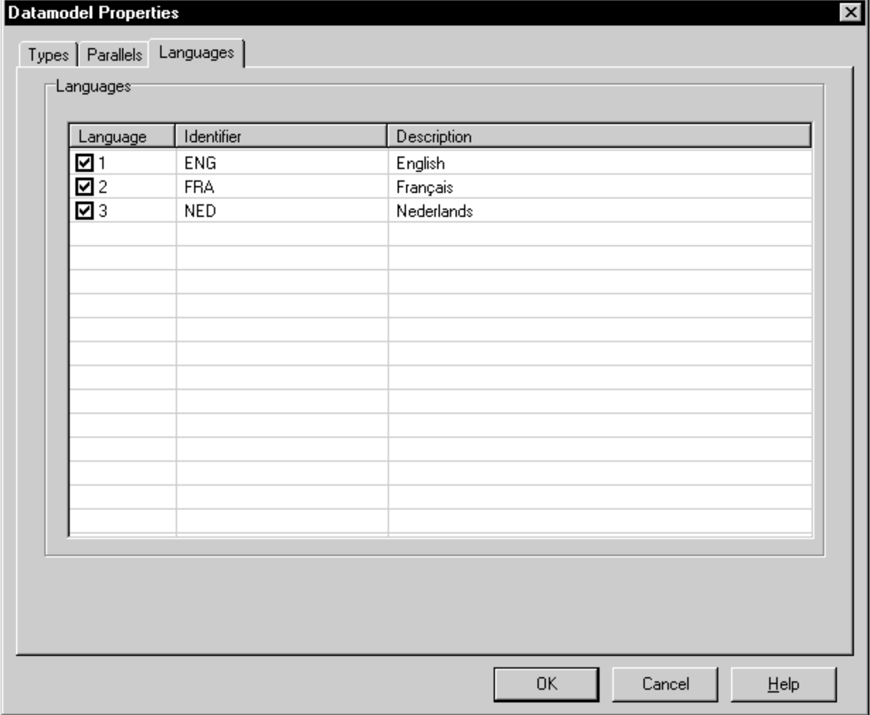
For more details on parallel blocks, see Chapter 4.

6.6.3 Languages properties

On this tab you can specify which of the defined languages needs to be accessible for the interviewer in the DEP. These languages are accessible in the language dialog or via the previous and next language commands in the DEP. You can select a language for the DEP by checking the appropriate check box in front of the language identifier in the lister you see on the screen. At least one language needs to be checked.

Only the accessible languages can be activated via the command line parameter /L. The execution of the DEP will fail if you try to select a language via the command line that is not accessible.

Figure 6-47: Datamodel properties—Languages

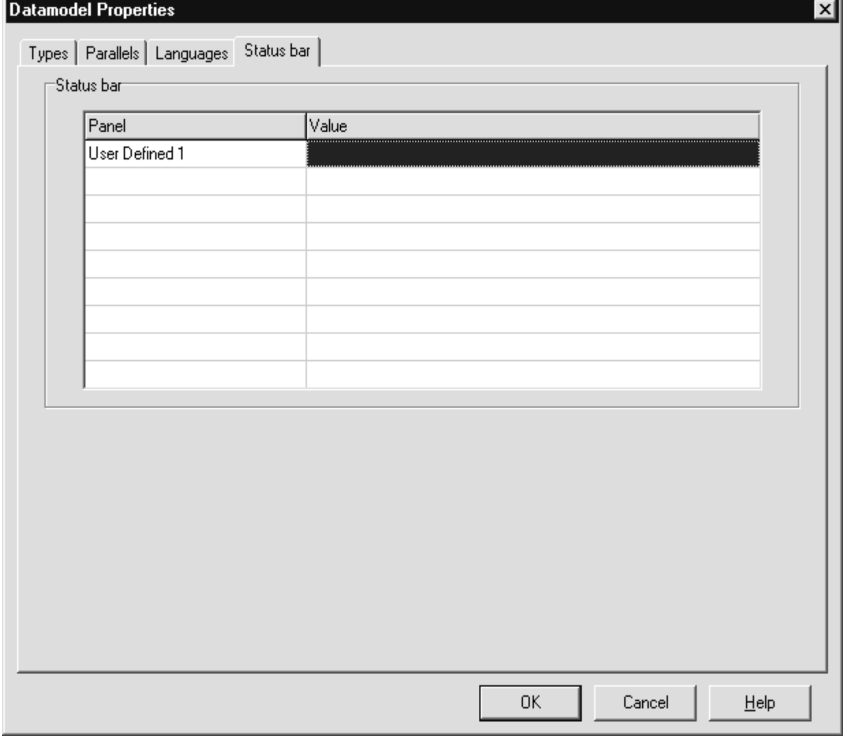


6.6.4 Status bar properties

On this tab you can specify the text for the user defined status bar panels. This tab is only visible if you have specified in the mode library that you want to display user defined panels on the status bar.

For each user-defined panel you can specify a text in the Value column. This text may contain fills. The fill must be based on a fully qualified field name. A possible subscript in such a name must be a numerical constant. The fills will be replaced run-time by the actual current value.

Figure 6-48: Data model properties—Status bar



! The system does not check the fills specified in the text. This is done by the DEP. If the field can not be found in the data model, the fill will be skipped.

6.7 DEP Configuration File

The DEP configuration file is a customisation file that can be used for DEP font and colour enhancements and behaviour toggles. These same settings can also be set in the modelib file, but any settings in the DEP configuration file will override the settings stored in the DEP modes file (`.bdm`). Using the DEP configuration file means you can change aspects of the instrument's behaviour without re-preparing or replacing instrument files.

The DEP Configuration Program (or DEPCfg, for short) is the Blaise tool that you use to create and edit a DEP configuration file. The DEP configuration file

has a `.diw` file extension. The DEPCfg Program works in conjunction with the Mode Library Editor. You can use the Mode Library Editor to set up a modelib file, and you can then use DEPCfg to create a `.diw` file to override the modelib settings for specific data models. You can override the *Style* and *Toggle* settings, and the *Layout* colour settings. You cannot set any other *Layout* settings in the DEPCfg Program.

To work on a configuration file, you must open a data model. This reflects that idea that a DEP configuration file is based upon a data model, or a class of data models, prepared with the same mode library file.

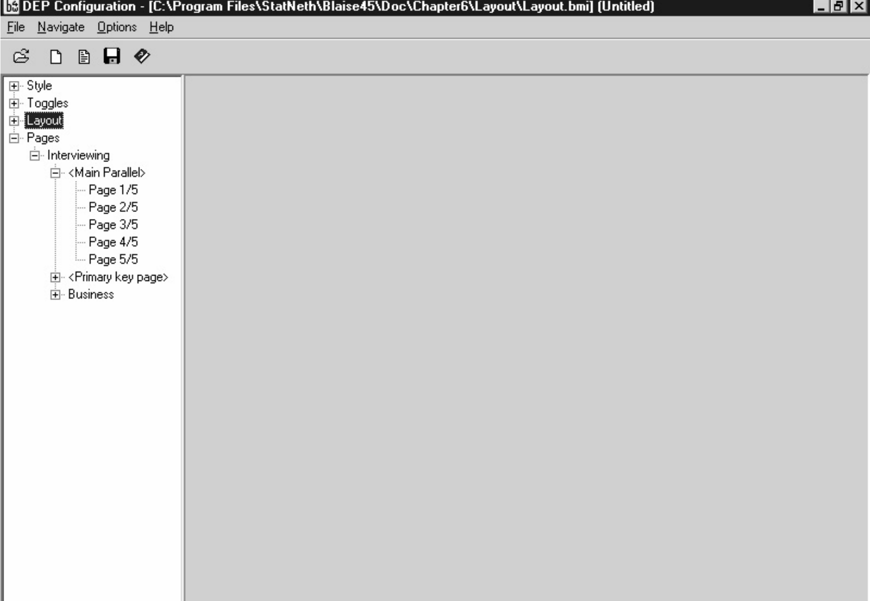
6.7.1 Using the DEP Configuration Program

This section describes how to open, save, and set options in the DEP Configuration Program.

Open the DEP Configuration Program

To open the DEP Configuration program, select *Tools ➤ DEP Configuration* from the Control Centre menu. The *DEP Configuration* window appears.

Figure 6-49: DEP Configuration Program



This program looks very similar to the Mode Library Editor, and, in fact, has many of the same features.

Open or create a configuration file

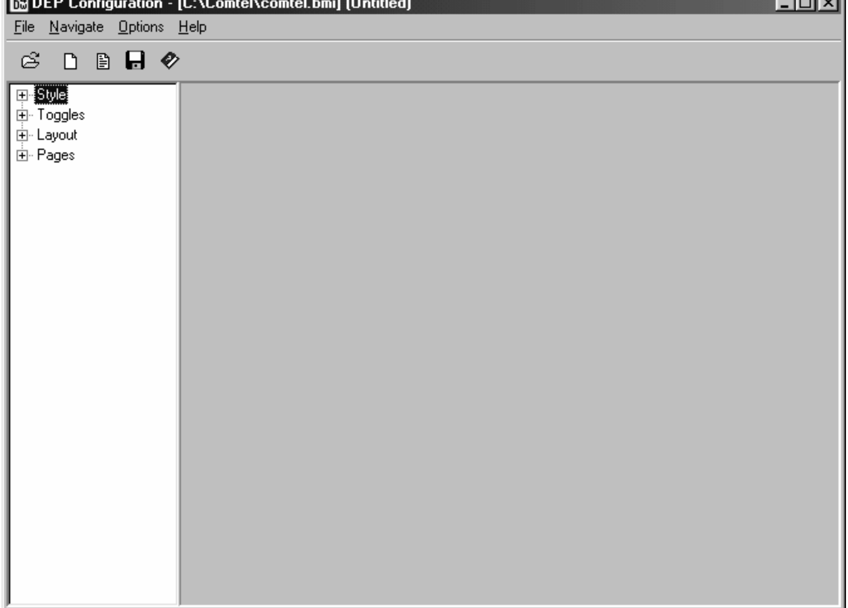
To work on a DEP configuration file, you must first open a data model in the DEP Configuration Program. Opening the data model allows DEPCfg to read the modelib settings.

Select *File* ➢ *Open data model*, and select a data model's `.bmi` file. A branch called *Pages* appears in the tree view on the left.

Open or create a configuration file by selecting *File* ➢ *Open Configuration File* or *New Configuration File*. If you are opening a file, select a file with a `.diw` extension.

The tree view then displays *Style*, *Toggles*, *Layout,* and *Pages* branches, as shown in the following figure:

Figure 6-50: Creating a .diw file in the DEPCfg Program

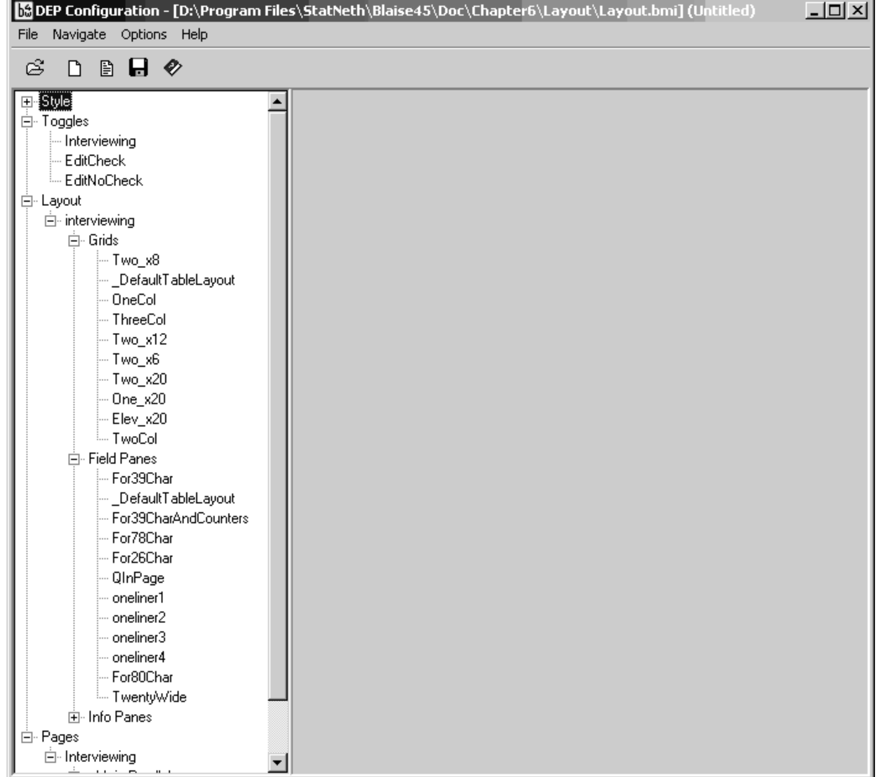


- The *Style* branch defines fonts, colour schemes, and DEP options.
- The *Toggles* branch defines behaviours.

- The *Layout* branch shows the settings for the size and appearance of Grids, FieldPanes, and InfoPanes.
- The *Pages* branch shows the effects on the forms of any changes to the *Style* or *Layout*.

Expand the branches on the tree view on the left. The behaviour identifiers that you see in the *Toggles* branch are the identifiers that were created in the modelib file that the data model was prepared under. The same is true of the *Layout* branch. It displays the layout sets for that data model's modelib file, as shown in the following figure:

Figure 6-51: Layout identifiers in the DEPCfg Program



Save the configuration file

Once all changes have been made, save the configuration file by selecting *File ➢ Save.*

Meta search path option

If you open a data model that uses other data models, you can set an option to have the program search for the meta files of the other data models. Select *Options ➢ Environment Options*, and type in a search path.

6.7.2 Editing a DEP configuration file

The settings for *Style* and *Toggles* in the DEP Configuration Program are identical to the settings in the Mode Library Editor.

- You can change all *Style* settings, including fonts, options, and colour schemes.
- You can change and edit all *Toggles*. If you want behaviours in the DEP configuration file to override the behaviours in the modelib file, the behaviour identifiers (the names you specify when you add a toggle set) must exactly match the behaviour identifier names in the modelib file. If the names are different, the behaviours will not be overridden.
- The only layout settings you can edit in the DEPCfg Program are colours. You can view the layout settings, but you cannot change them.
- You can also view pages in the DEPCfg Program.

Refer to the appropriate sections under section *6.5 Mode Library File* for specific descriptions of *Style, Toggles, Layout,* and *Pages.*

6.7.3 Applying a DEP configuration file

There are a few ways to apply the configuration file to your data model. Remember that all settings in the `.diw` file will override the corresponding settings in the modelib file when the `.diw` is applied.

Apply to all data models

To use the `.diw` file for all data models run from the Control Centre, specify the file name in the run parameters. This will apply the configuration file to each data model that is run from the Control Centre.

Apply to specific data models

To apply a `.diw` file to just one data model, use the command line parameter */C* when you invoke the `dep.exe` command to run the DEP. For example, to run the

data model `ncs07.bla` and apply the file `ncs.diw`, you would use the command:

```
DEP NCS07 /CNCS.diw
```

As another example, you could apply different `.diw` files to the same data model to achieve different behaviour effects. For example, if you had defined two `.diw` files, `ncsint.diw` for interviewing and `ncsedit.diw` for data editing, you could run the DEP for the data model `ncs97.bla` with the command line parameters:

```
DEP NCS97 /CNCSInt.diw /T1 /P1     (for interviwers)
```

```
DEP NCS97 /CNCSEdit.diw /T2 /P2    (for data editors)
```

In this example, /T1 and /P1 point to the first set of layouts and behaviours in the modelib file, which are for the interviewer. The /T2 and /P2 parameters point to the second set of layouts and behaviours in the modelib file, which are for the editor.

The command line parameters could be associated with a Windows® shortcut, a Maniplus command, or a batch file. See Appendix A for a list of all command line parameters.

6.8 Menu File and the DEP Menu Manager

The DEP menu file controls the menu and speed buttons available in the DEP. With this file you can enable or disable a menu item, clear a submenu from the menu altogether, assign a function key to a menu item, assign shortcut keys to a menu command, and assign speed buttons to the Speedbar. You can also define a new menu item, and link the menu item to an executable, a DLL procedure, or a parallel block.

There are several menu files provided with the Blaise system. `Depmenu.bwm` is the default menu file. For CATI surveys, `Catimenu.bwm` is the default. There is a third file provided, `Capimenu.bwm`. You can accept the default file, use one of the other files, or edit the default file and apply it to different data models.

The *Edit|Update* is an option that can be used to update a menu file to reflect the current menu of the DEP. The option will also change the menu texts (caption and hint) to the currently active system language. *Edit|Update* is available after you have loaded a menu file and as long as no changes have been made to the file. All user defined menu entries remain present.

For example, if you have an old menu file and you want to make use of the new auto dial menu entry you can choose the *Edit|Update* option. The auto dial option (and the new Send option) will be added to the menu in the tree view.

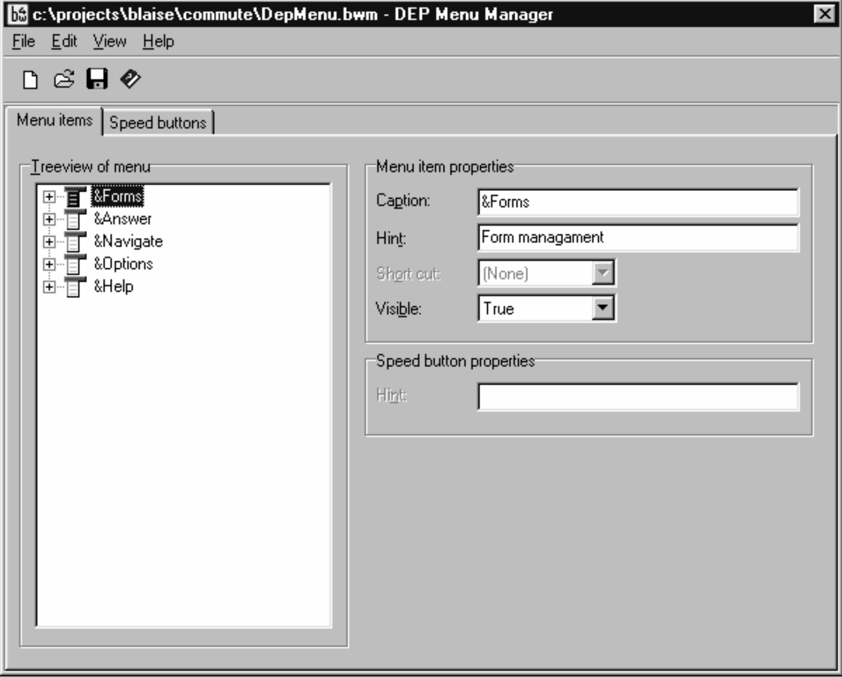
If you edit the default menu file, save it to a new file name so that you can revert to or use the default settings when necessary.

6.8.1 Using the DEP Menu Manager

To open the DEP Menu Manager, select *Tools ➢ DEP Menu Manager* from the Control Centre menu. The *DEP Menu Manager* window appears. It is a blank window with several menu options.

To open or create a menu file, select *File ➢ Open* from the menu, and select a menu file with a `.bwm` extension. Create a new file by selecting *File ➢ New.* A tree view of the menu file appears on the left, and properties of each menu item are on the right.

Figure 6-52: DEP Menu Manager



There are two tabsheets: *Menu items* and *Speed buttons.* These are described in the following sections.

To save your file, select *File* ➢ *Save* from the menu.

6.8.2 Editing and adding menu items

On the *Menu items* tab, the *Treeview of menu* box contains the default menu options *Forms, Answer, Navigate, Options,* and *Help*. Expand and collapse the menu items by clicking the plus or minus sign that appears to the left of the text. Menu items that have a red X next to them are currently not visible in the DEP.

Editing menu items

To edit the existing menu items, select the menu item to be edited, and complete the menu item properties as described as follows:

In the *Menu item properties* section:

- *Caption.* Type the caption that you want displayed for the menu item. To set a shortcut key for one of the letters, insert an ampersand (&) before the letter.
- *Hint.* Type the words that you want to appear in the status bar of the DEP screen when the menu option is selected.
- *Shortcut.* Select a function key or short-cut key from the drop down list.
- *Visible.* Select *True* to make the menu option visible in the DEP; select *False* to make the option not visible. When you choose *False,* a red X appears on the menu option in the tree. You can also double click on the menu in the tree to toggle the visibility on and off. If you choose *False* for a top-level menu, all submenu options under the menu item will also be invisible. If you then make the top-level menu visible again, you must make each submenu item visible again individually.

In the *Speed button properties* section:

- *Hint.* If the menu item has a corresponding speed button on the *Speed buttons* tab, specify the text that should appear when the cursor is placed on the speed button.

Adding a user-defined menu entry

You can add a user-defined menu entry under one of the predefined main menu items or to a user-defined main menu item. User-defined menu items can invoke a DLL procedure, start an executable, invoke a COM object method, invoke an advanced DLL procedure, or start a parallel block. In order to invoke a COM or advanced DLL procedure you must have the Blaise Component Pack installed.

Place the cursor at the location in the tree view where you want to add a new entry. From the menu, select *Edit ➢ Add Menu Entry*. A space for the new menu item appears above the cursor.

Move the new menu item up and down within the submenu by pressing *F7* and *Shift-F7*. You can also right click on the tree view to add, delete, or move menu items.

Complete the menu item properties as described above. User-defined menu entries have some extra properties that can be set, which are described as follows:

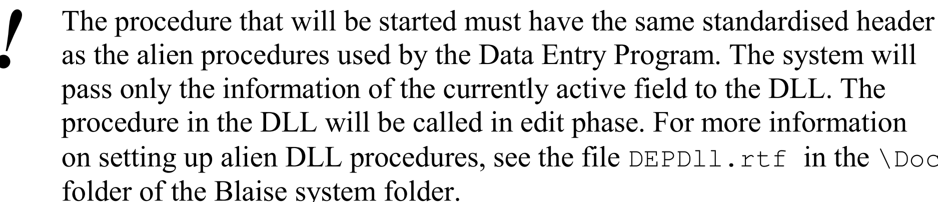
- *Identifier.* If you want to link a WinHelp item to the menu entry, specify the identifier. If you leave this field empty, no help topic will be linked to the menu entry.

- *Kind.* Select the kind of action to be performed by the menu entry: *Invoke a DLL procedure, Start executable, Start parallel Invoke a DLL procedure (advanced),* or *Invoke COM object method.* Each of these menu entry types has its own properties, which are described in the following sections.

Menu properties for starting a DLL procedure

For menus that start a DLL procedure, the following properties can be set:

- *DLL name.* Specify the name of the DLL that contains the procedure you want to invoke. You can type the DLL name or browse for it.
- *Procedure name.* Specify the name of the procedure that you want to invoke. If the name of the DLL is specified, the drop down list will contain all available procedures in the DLL. If you type the name of the DLL yourself, be sure to specify the correct case for each character of the procedure name.



The procedure that will be started must have the same standardised header as the alien procedures used by the Data Entry Program. The system will pass only the information of the currently active field to the DLL. The procedure in the DLL will be called in edit phase. For more information on setting up alien DLL procedures, see the file `DEPDll.rtf` in the `\Doc` folder of the Blaise system folder.

Menu properties for starting an advanced DLL procedure

For menus that start an advanced DLL procedure, the Blaise Component Pack must be installed. The following properties can be set:

- *DLL name.* Specify the name of the DLL that contains the procedure you want to invoke. You can type the DLL name or browse for it.
- *Procedure name.* Specify the name of the procedure that you want to invoke. If the name of the DLL is specified, the drop down list will contain all available procedures in the DLL. If you type the name of the DLL yourself, be sure to specify the correct case for each character of the procedure name. In addition, the procedure must have a standardised header with two parameters: *IBlaiseDatabase* and *IDepStatus*. Please read the help provided with the Blaise Component Pack for more information on these two parameters.

Menu properties to invoke a COM object method

For menus that invoke an advanced COM Object Method, the Blaise Component Pack must be installed. The following properties can be set:

- *Prog ID.* Specify the ProgID that contains the method you want to invoke. Be sure to write the correct case for each character of the ProgID.
- *Method name.* Specify the name of the method that you want to invoke. Be sure to write the correct case for each character of the method name. The procedure that will be started must have a standardised header with two parameters: IBlaiseDatabase and IDepStatus. For more detailed information, refer to the available help in the Blaise Component Pack.

Menu properties for starting an executable

For menus that start an executable, the following properties can be set:

- *Command.* Specify the name of the executable plus the command line parameters. You can use the following macros in the command:

Figure 6-53: List of command macros

$FieldName	Will be replaced by the local name of the current active field in the DEP when you choose this menu entry.
$FieldTag	Will be replaced by the local tag name of the current active field in the DEP when you choose this menu entry.
$FieldValue	Will be replaced by the value of the current active field in the DEP when you choose this menu entry
$Value(FieldName)	Will be replaced by the value of the field *FieldName* when you choose this menu entry.
$DataName	Will be replaced by the name of the current active database in the DEP when you choose this menu entry.
$DictionaryName	Will be replaced by the name of the current active data model in the DEP when you choose this menu entry.
$DictionaryVersion	Will be replaced by the version information of the current database. This version information is the version information of the dictionary when the database was created. The version information has the following format: Ma.Mi.Rel.Bn. Ma = Major version number, Mi = Minor version number, Rel = Release number, Bn = Build number. Version information is specified in Project Options. See section 2.2.6
$DataVersion.	Will be replaced by the version info of the current database. This version info is the version info of the dictionary when the database was created. The version info has the following format: Ma.Mi.Rel.Bn. Ma = Major version number, Mi = Minor version number, Rel = Release number, Bn = Build number. See section 2.2.6
$Primary	Will be replaced by the value of the primary key of the current form (if available) when you choose this menu entry. The format of the primary key is the same as the format of key the DEP accepts on the command line (/K option). If a space is present in the primary key, it will be placed between double quotes.

If the value of a macro cannot be determined, it will be removed (replaced by empty).

- *Wait until finished.* Select to have the DEP wait (DEP will not respond at all) until the program that has been started has finished.
- *Minimise on run.* Select to have the DEP window minimise when you choose this menu entry.

Menu properties for starting a parallel block

For menus that start a parallel, the following property can be set:

- *Parallel.* Specify the name of the parallel block that you want to start. If you leave this property empty, the main parallel will be attached to this menu item. When you choose this menu entry, the DEP will search for a parallel with the specified name. If the parallel is not found, or the parallel is currently not accessible, the menu entry in the DEP will be greyed out.

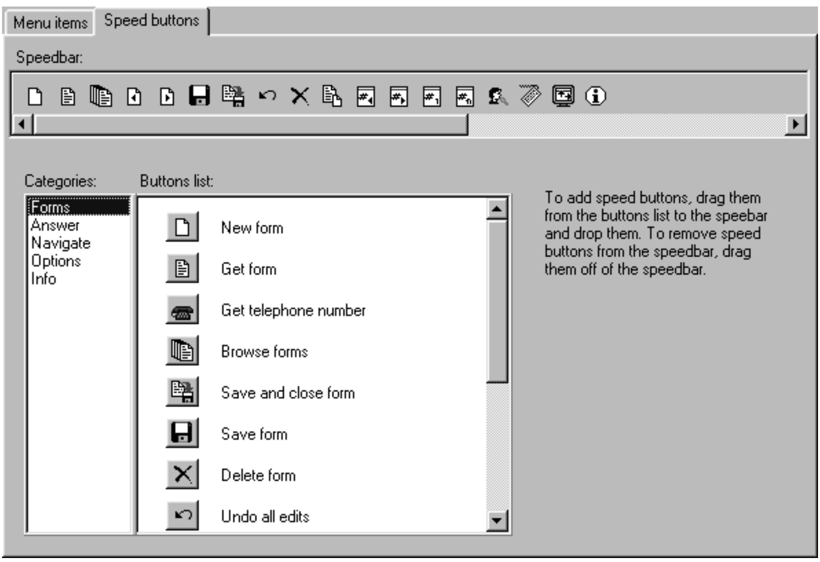
Delete a user-defined menu entry

To delete a user-defined menu entry, select the entry you want to remove in the tree view, and select *Edit* ➤ *Delete Menu Entry*. You can also right click on the menu item, and select from the pop-up menu.

6.8.3 Editing and adding speed buttons

On the *Speed buttons* tabsheet, there is a list of speed buttons currently available in the menu file, as shown in Figure 6-49.

Figure 6-54: Speed buttons tabsheet in the DEP Menu Manager



Select a category on the left to see the available buttons.

To add speed buttons, drag them from the *Buttons list* to the *Speedbar* box. To remove buttons, drag them from the *Speedbar*.

A note about CATI menu files

If you develop a menu file for a CATI survey you must use a CATI menu file. You need the menu option *Get telephone number* if you want forms to be delivered automatically to the interviewer. The menu options *Get* and *Browse* are also suggested. The *Get* option allows the interviewer to request a form by a primary key. The *Browse* option allows interviewers to browse through all the forms in the data file, based on either a primary or a secondary key.

There is a CATI menu file, `catimenu.bwm`, in the Blaise system folder. By default, the system uses the `catimenu.bwm` file when running a CATI instrument. You can use this menu or edit it to suit your needs. See Chapter 10 for more information on using CATI menus.

6.8.4 Applying a menu file

Apply to all data models

To use the menu file for all data models run from the Control Centre, specify the file name in the DEP run parameters. This will cause each data model that is run from the Control Centre to use that file.

Apply to specific data models

To apply a menu file to a specific data model, use the command line parameter */M* when you run the `dep.exe` command. For example, to run the data model `ncs07.bla` and apply the menu file `ncs.bwm`, you would use the command:

```
DEP NCS07 /Mncs.bwm
```

6.9 Screen Layout Considerations

It is important to keep in mind that screen layout is a combination of many factors.

6.9.1 Data density in the page

The density of data in the FormPane (page) is determined by the Grid definition. Increased data density is popular with interviewers for several reasons. They get a good overview of the instrument. They can see the routing unfold as data are entered. They can see the responses to several or many previous questions. The arrow keys or mouse are very effective for navigating within the page. The page up and page down keys are extremely effective for navigating between pages. Though it is possible to find situations where another screen style is needed, the default screen presentation is very effective for many applications.

6.9.2 Font sizes

The measures used in the mode library file for the number of rows for the vertical dimension and the number of characters for the horizontal dimension are based on pixels. A scaling factor is calculated based on the font size set in the DEP configuration file.

One font setting that is extremely important is the font type and size set in the *Style* setting of the Mode Library Editor. By changing this setting, you can allow

space for many more or many fewer screen elements, such as the number of fields. Depending on the resolution and size of the screen, you can increase the number of fields and other screen elements. You should experiment with various combinations of mode library style definitions.

6.9.3 New pages created for new Grids

Every time a new Grid definition is applied, a new page is generated in the DEP. Thus if you move the horizontal dividing line up or down from one field to the next, the subsequent fields will be placed on a new page.

6.9.4 Screen resolution

Screen resolution varies from one computer to another. An application prepared on one computer can take up a smaller or larger amount of screen space on another computer. There are two ways to provide for this. One is to have a separate DEP configuration file, with style, tab, font, type, and size, for each possible screen resolution. Another way to accommodate these differences is to prepare the application on a computer with a low screen resolution. Moving up to a higher resolution screen is automatic, but the reverse is not true.

6.9.5 Summary of screen layout factors

The following summarises some the screen layout factors to be considered when designing DEP window presentations.

LAYOUT section

In the LAYOUT section of the data model, you refer to the specific possibilities that are in the modelib file and prepare the model using that modelib file. If you don’t have a LAYOUT section in the data model, it will use the first layout possibility in the modelib file.

Modelib settings

The modelib settings affect the size and appearance of the FormPane, InfoPane, FieldPane, and Grid, and DEP behaviours. Be sure to prepare the data model using the correct modelib file, and that all settings work together.

Configuration file

Configuration file settings override behaviour, text, and style settings that were set in the modelib file.

Hardware

Keep in mind that the type of computer used for development is not likely to be the same type of computer used when the instrument is run. Differences in monitor size and resolution will affect how an instrument appears.

Windows settings

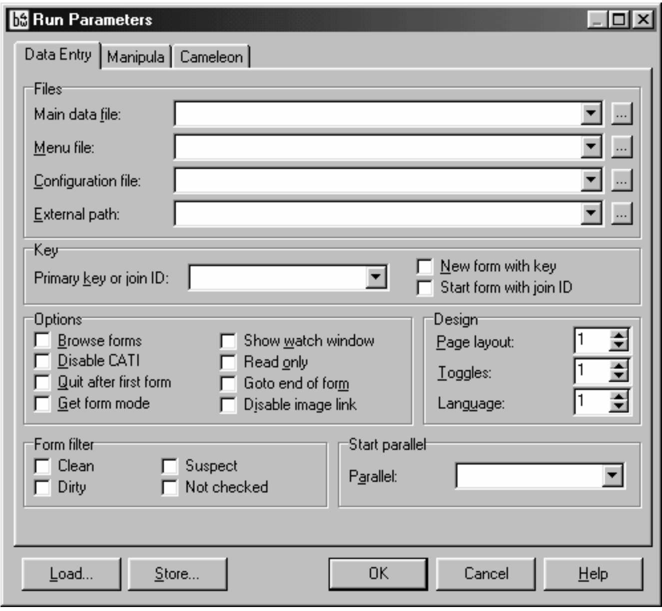
Blaise will use the Windows® regional settings for time and for date, if no other format is specified under the data model properties in the Control Centre. For more information, see Chapter 6, section *6.6 Data model properties*.

DEP Run Parameters

When you run the DEP from the Control Centre, you can set Run parameters. This is one way to apply different configuration and menu files to data models while you develop and test.

To set run parameters, open the Control Centre. Select *Run ➤ Parameters* from the menu and the *Run Parameters* dialog box appears. Select the *Data Entry* tab and set parameters as described in Figure 6-55.

Figure 6-55: DEP Run Parameters



In the *Files* section:

- *Main data file.* Specify the path and name for the data file. If left blank, the data file will have the same name as the data model.
- *Menu file.* Specify the path and file name of the menu file to use. If left blank, Blaise uses the default menu file `depmenu.bwm`, or for CATI instruments, `catimenu.bwm`.
- *Configuration file.* Specify the path and file name of a DEP configuration file to use.
- *External path.* If the data model uses an external file, type the path to use for the external data file.

Key

In the *Key* section:

- Specify a form to retrieve using the value of its primary key (as defined in the data model) or join ID (its internal record number). Specify a value in the *Primary key or join ID* box, then click the *New Form with key* box to retrieve on primary key, or the *Start form with join ID* box to retrieve on the join ID. This will cause that form to open when the DEP runs.

Options

In the *Options* section:

- *Browse forms.* Select to allow users to scroll through all the forms and select a specific one when the DEP is run.
- *Disable CATI.* Select to run the DEP in interviewing mode. Use this option if your instrument contains the phrase INHERIT CATI and you want to run the DEP without using the call scheduler.
- *Quit after first form.* Select to cause the DEP to close after one form is completed.
- *Get form mode.* Select to run the DEP in *Get form* mode, causing the interviewer to retrieve existing forms.
- *Show watch window.* Instruct the DEP to activate the watch window. The watch window can be used to display the values of fields and auxfields in your data model and it can display which blocks have been checked and which external files have been access.
- *Read only.* Read-only mode for forms. When set all existing forms in the file can be read, but not modified or saved.
- *Go to end of form.* Go directly to last field on the route that needs to be answered. This option is similar to pressing the END key directly after loading a form.
- *Disable image link.* Disable Image link of the DEP if necessary. Blaise can show images of scanned forms in a Blaise data entry session. This functionality is only available when the *Blaise Component Pack* is installed and is enabled by using the keyword, *INHERIT* followed by *IMGLINK*. *INHERIT IMGLINK* includes a special block with fields that are necessary for viewing scanned forms. For more information on Image links, see the on-line help topics *Showing form images in the DEP*.

Design

In the *Design* section:

- *Page layout.* Specify the number of the layout to be used from the modelib file.
- *Toggles.* Specify the number of the behaviour toggle from the configuration file.
- *Language.* Specify a language number to be used, as specified in the data model.

Form filter

In the *Form filter* section one may specify which forms will be presented using the Dep menu selections Forms > Next Form (F7) or Forms > Previous Form (Alt-F7). The filter settings don't apply for menu selection Browse forms. The settings are:

- *Clean.* Include clean forms in the form read filter.
- *Dirty.* Include dirty forms in the form read filter.
- *Suspect.* Include suspect forms in the form read filter.
- *Not checked.* Include not checked forms in the form read filter.

Start parallel

In the *Start parallel* section:

- *Start parallel.* Go directly to the parallel specified when entering the form. This option will only work when the key page is not activated when starting the DEP and when the specified parallel can be reached.

Load and Store buttons

You can use the load button to read the command line parameters from a Blaise command line option file. You can use the store button to write the command line parameters to a Blaise command line option file.

6.10 Using the DEP

There are many ways to customise the look and behaviour of the DEP and it would be impossible to describe how to use the DEP for all possible combinations

of layouts and behaviours. Here we describe some of the major features of using the DEP and show samples of typical settings.

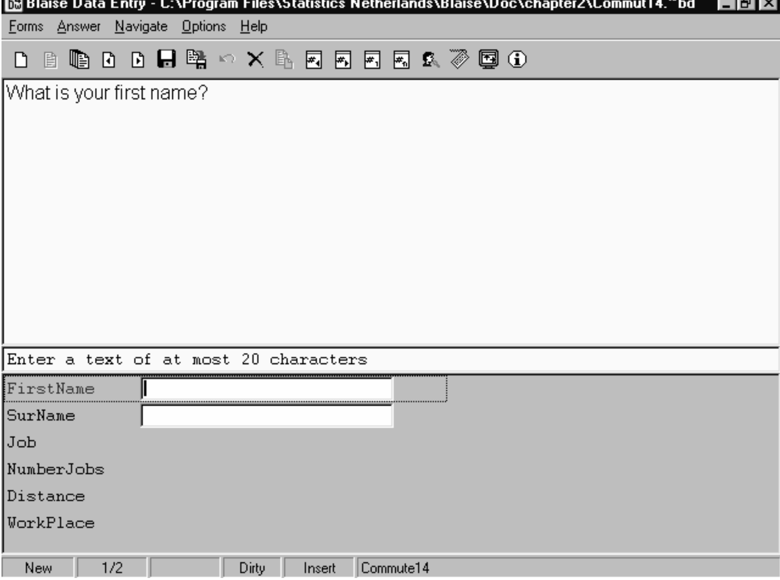
6.10.1 Invoking a behaviour mode: interviewing or data editing

The modes of behaviour are determined by settings in the modelib file, and how those modes behave is determined in the configuration file. When you run the DEP, the default mode of behaviour is automatically invoked.

To select a mode, select *Options ➢ Data Entry Mode* from the menu. You then have a choice of *Interviewing*, *Data Editing*, or *Self Defined*. The exact behaviours associated with each of these options is determined by the developer in the configuration file.

The following is a sample of `commut14.bla` being run in Interviewing mode. (`Commut14.bla` can be found in `\Doc\Chapter4` of the Blaise system folder.)

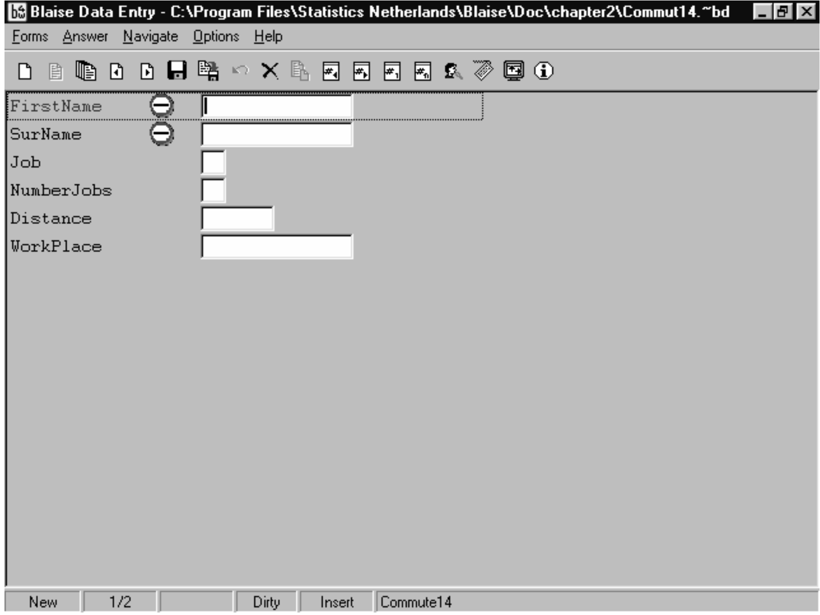
Figure 6-56: DEP in interviewing mode



This window is split into two parts: the InfoPane is in the top half and contains the question text for the interviewer. The FormPane is in the bottom half and contains the fields. Here you are really using a behaviour and a layout together. The behaviour is dynamic routing, checking, and display, with a split-screen layout.

The following sample shows the same data model being run in typical data editing mode.

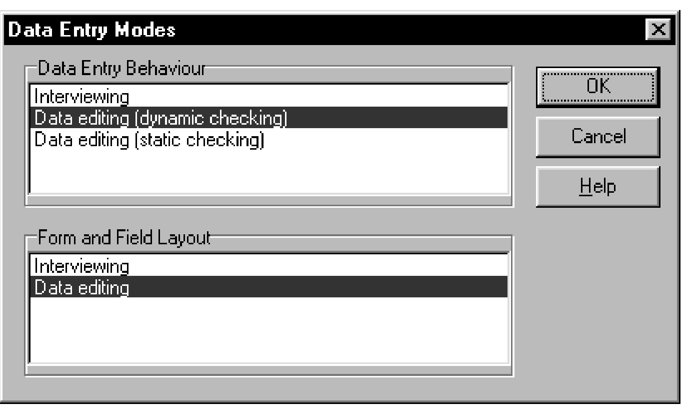
Figure 6-57: DEP in data editing mode



In this mode, there is no InfoPane with information for an interviewer. In fact, the question text is not displayed at all. The symbols next to the first two fields are error symbols, which are described later in this chapter.

If *Self Defined* mode has been allowed by the developer, the *Data Entry Modes* dialog box appears with a list of customised specification labels.

Figure 6-58: Data Entry Modes dialog box



The available choices are again defined by the developer. Select the *Data Entry Behaviour* and the *Form and Field Layout* that you want and click the *OK* button.

The data entry behaviour is independent of the form and field layout. For example, you can choose a data editing behaviour and still maintain an interviewing or self-defined layout. It is also possible to have a data editor see question text but still have navigational capability that the interviewer doesn't have.

Allowing self-defined behaviour should be done with care. Allowing interviewers to select a checking behaviour could affect the quality of the data collection, but data editors may have reasons to switch modes.

6.10.2 Entering responses

When entering data into a Blaise instrument, you typically type responses in the appropriate boxes and, if necessary, press the Enter key to move to the next field. At the end of the form, you might be prompted to save the form. This is true in both Interviewing and Data Editing modes.

In production, users will most likely use shortcut keys to access functions and menu commands. Menu commands are not often used in production, mostly to save time. Because you can set your own shortcut keys, in this section we have specified the menu command to access the functions.

For reference, a table of the default shortcut keys is in the following figure:

Figure 6-59: Default shortcut keys in the DEP

Shortcut Key	Description
Alt-F2	New form
Ctrl-F1	Question help
Ctrl-F2	Delete form
Ctrl-F7	Browse forms
Ctrl-F9	Undo all edits
Ctrl-M	Make remark
Ctrl-S	Sub forms
Ctrl-T	Ditto
End	Last page in the form
F1	Help
F2	Switch between insert and navigate mode on the input line
F4	Errors in field
F7	Next form
F9	Show question text
Home	First page in the form
PageDown	Next page in the form
PageUp	Previous page in the form
Shift-F2	Save form
Shift-F7	Previous form
Shift-F9	Search tag

Don’t know and refusal

To record a *Don’t Know* or *Refuse* response, select these from the *Answer* menu, or use the shortcut keys. When you do so, a symbol appears in the field.

Don’t Know is a question mark. ?

Refuse is an exclamation point. !

Remarks

To record a remark, select *Answer* ➢ *Remark* from the menu. The *Remark* dialog box displays.

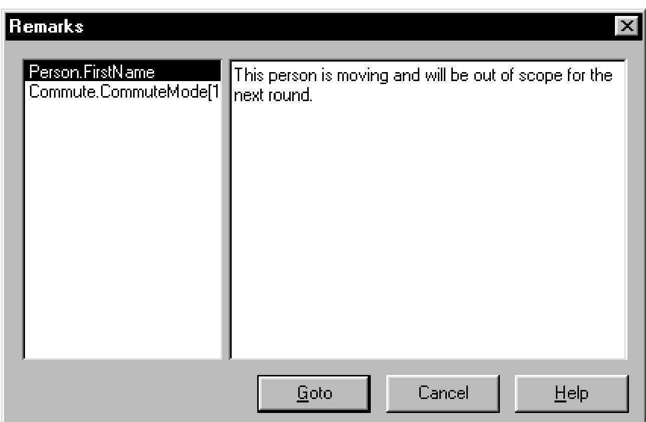
Figure 6-60: Remark dialog box

Type a remark and then click the *Save* button. A small paperclip symbol appears next to the field. To view the remark, select *Answer* ➢ *Remark* from the menu, or double-click the symbol.

- *Ditto*. The *Answer* ➢ *Ditto* command copies the contents of a field from the previous form into the same field in the current form.
- *Repeat.* The *Answer* ➢ *Repeat* command copies the contents of the response of the nearest field (that was earlier on the route) that has the same local name on the current page. *Local name* refers to the name of the field without the surrounding block name. This option is useful in tables, where each row is normally of the same block type. In this case, all cells in a column have the same local name, and this command copies the value of the cell above.
- *Show question text.* The menu option *Answer* ➢ *Show question text* displays a dialog box with the question text. This might be useful in data editing mode, when you probably won’t have an InfoPane with the question text. This is also used for help text.
- *Show all remarks, open answers, don’t knows, refusals*. To view all remarks, open answers, don’t know responses, and refusals in the current form, select

the appropriate option from the *Navigate* menu and a dialog box appears. The sample below is the *Remarks* dialog box.

Figure 6-61: Showing remarks in the DEP

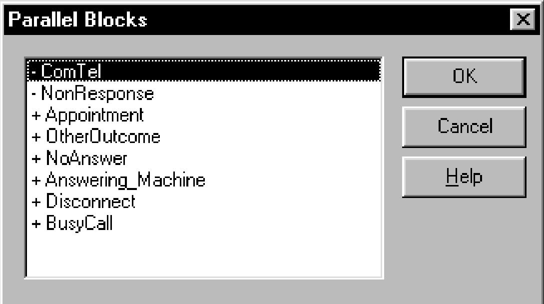


The fields that contain remarks are on the left, and the actual remark text is on the right. Select a field using the up and down arrows or by clicking it in the left column. To go to the field, select it and either press *Enter* or click the *Goto* button.

Sub forms (accessing parallel blocks)

Parallel blocks in a data model allow you to break out of the order specified by the rules and go directly to a different block. If the data model has parallel blocks, access those blocks by selecting *Navigate ➢ Sub Forms* from the menu. The *Parallel Blocks* dialog box appears.

Figure 6-62: Parallel Blocks dialog box



Select the block you want to go to and click the *OK* button. You can also quit the current form from this box.

If the parallel blocks appear on a tabsheet, access the block by selecting the appropriate tabsheet. You may also operate the tabs used for the parallels with the keyboard. The normal key combination under Windows® to focus the next/previous tab is (Shift)-Ctrl-Tab. Because Ctrl-Tab is also reserved to switch between the form pane and the info pane, some changes have been made. In the case when no tabs are present you can still use Ctrl-Tab to switch between the form pane and the info pane. You can now also use the F6 key for this. In the case when tabs are present, Ctrl-Tab (and Shift-Ctrl-Tab) can be used to focus the next/previous tab and F6 needs to be used to switch between the form pane and the info pane.

You can also start a parallel block from a menu item. See “Menu properties for starting a parallel block” in section 6.8.2.

Start asker

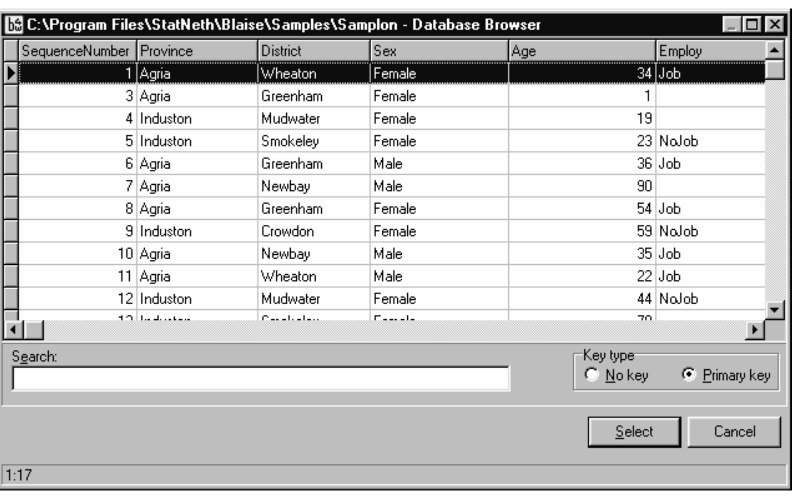
Use the menu command *Navigate ➢ Start asker* for fields that require a special action, such as a lookup. You can configure the DEP to require the user to use a short-cut key or menu command to perform the action, instead of having it occur automatically. If the DEP is configured in this way (it is a toggle in the DEP configuration file), you would use this command to perform the action.

6.10.3 Navigating between forms

There are several ways to access and move among forms in Blaise.

- *New form.* To create a new form, select *Forms ➢ New* from the menu. If you have not yet completed the current form, you will get a message prompting you to save the form before selecting a new one.
- *Get form by primary key.* If a primary key has been identified in the data model, access a specific form by its primary key value by selecting *Forms ➢ Get* from the menu. You *must* know the value of the key to access the form.
- *Browse forms.* To browse forms, select *Forms ➢ Browse* from the menu. All of the forms in the data file appear in the DEP window in the Database Browser.

Figure 6-63: Browse forms window



If keys are defined, you can browse on any key type.

Go to a form by selecting the form and pressing the Enter key or double clicking on the form. The form will then appear in the DEP window.

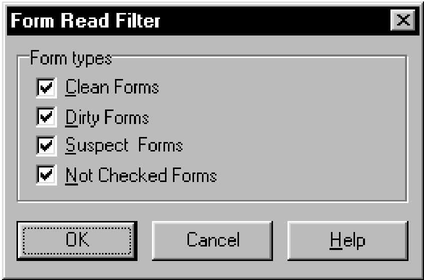
Close, save, delete

The *Forms ➢ Close*, *Save*, and *Delete* menu commands apply to the form currently displayed in the DEP. If you select *Close*, you are prompted to save the form. The *Delete* option is not usually allowed during production.

Form type

If you want to see only forms with a specific cleanliness status, then select *Forms ➢ Form type* from the menu. The *Form Read Filter* dialog box appears.

Figure 6-64: Form Read Filter dialog box



Select the type of forms you want to see and then click the *OK* button. See Figure 6-50. This might be used to review sets of forms. However, the Form Read Filter is only used when *Next form* or *Previous form* (see below) is used. It is not used when browsing forms.

Next and previous

To display the next or previous form, select *Forms* ➤ *Next form* or *Previous Form.* To see the next, previous, first, or last page of the current form, select *Navigate* ➤ *Next Page, Previous Page, First Page,* or *Last Page.* You can also use the keyboard keys *PageUp* (previous page), *PageDown* (next page), *Home* (first page), and *End* (last page). If the *Navigation* toolbar is visible, you can also use the icons on the bar to move between pages of the instrument.

Data file

You can open a data file in the DEP that was created using the same data model you are currently running. Select *Forms* ➤ *Data file* from the menu and select a Blaise data file. The DEP window reappears. Select to browse the forms, and the Database Browser opens with the data file.

Search tags

If a tag has been assigned to a field, you can search for the tag by selecting *Navigate* ➤ *Search tag* from the menu. The *Search Tag* dialog box appears.

Figure 6-65: Search Tag dialog box

Type the value of the tag that you are searching for, select to search from the first field in the form or from the current field, then click the *OK* button.

6.10.4 Errors

There are a few ways in which Blaise displays errors, and this can vary depending on which mode you are operating in.

Errors in interviewing mode

When you encounter an error in typical interviewing mode with dynamic error reporting, several things can happen. If you encounter a soft error, the *Active Signal* dialog box appears with the error message and the fields that caused the error.

Figure 6-66: Active signal dialog box

Active Signal

[WARNING E4] 120.0 minutes is possibly too many minutes!

Questions involved	Value
Commute.CommuteMode[1].UnitTime	Hours
Commute.CommuteMode[1].TotTime	2
Commute.CommuteMode[1].Minutes	120.0

Suppress | Close | Goto

You can either go to the error and correct it, or suppress it.

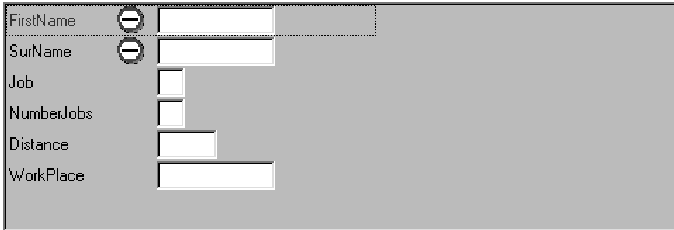
If you suppress a soft error, you have accepted the data, the form is then considered clean, and it is *not* labelled as suspect.

If you encounter a *hard* error, a *Hard Error* dialog box appears with the error message. This box looks exactly like the *Active Signal* box above, but you do not have the option to suppress the error; you must immediately correct it.

Errors in data editing mode

In typical data editing mode with static error reporting, you do not have to correct or attend to errors as they occur. When an error occurs, an icon or a number appears next to the field that has the error, which is determined in the DEP configuration file. Figure 6-62 shows the error icons.

Figure 6-67: Error icons in data editing mode



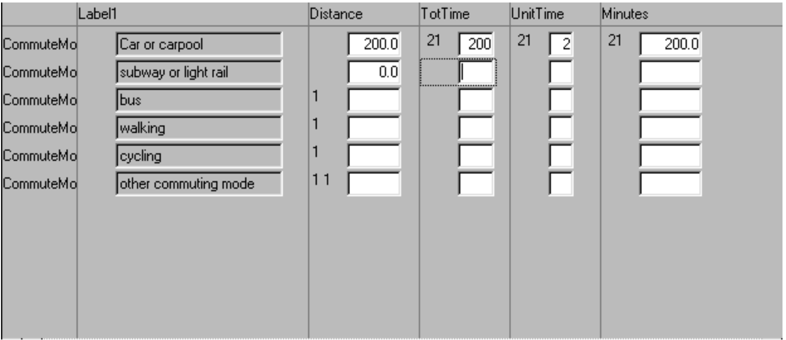
The following table gives the meaning of the icons.

Figure 6-68: Error icon definitions

Icon	Description
	Route error
	Hard error
	Soft error

If, however, the option *Show error counters* is selected in the mode library file or the DEP configuration file that was used for the data model, error counters will appear instead of icons. Up to three counters can appear next to a cell: one counter for route errors, one for hard errors, and one for soft errors. The number of each type of error in each field appears next to it.

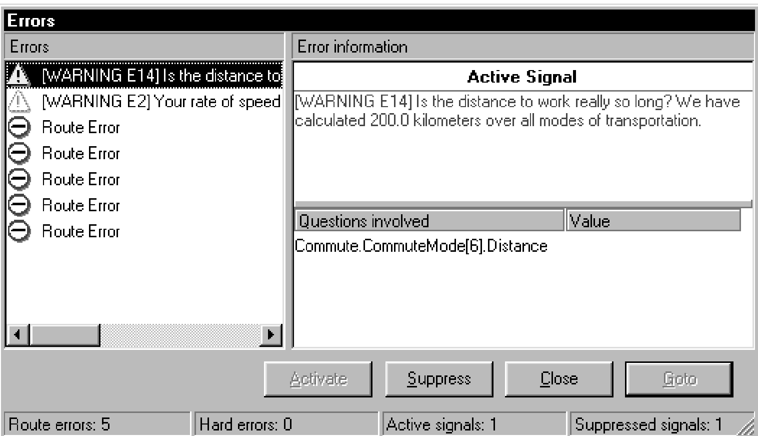
Figure 6-69: Error counters in data editing mode



Show errors

To see all errors in the current field, select *Navigate ➤ Errors in Field.* To see all errors in the current form, select *Navigate ➤ Show All Errors.* When you choose either of these two options, the *Errors* dialog box appears.

Figure 6-70: Errors dialog box



The errors are listed on the left and error information is on the right. To go to an error, click on the field in the *Questions involved* section in the lower right corner, and then click the *Goto* button.

To see only specific types of errors, select *Navigate ➤ Select Error Types.* A dialog box appears and you can choose to see route errors, hard errors, active signals, and suppressed signals. When you return to the DEP, the error symbols still display, but when you want to look at errors, only the selected types display.

6.10.5 Languages

If you specify language options in your data model, you can easily switch between language modes when running the DEP.

To see all available language options, select *Options ➤ Form Language* from the menu. The *Form Languages* dialog box appears. You could also set up function keys to toggle between languages.

Figure 6-71: Form Languages dialog box

Form Languages

Name	Description
ENGLISH	English
FRENCH	French
SPANISH	Spanish

OK
Cancel
Help

All spoken languages available in the current data model are listed. Select a language and click the *OK* button. The DEP will then use the selected language for all questions. You can also toggle between languages by selecting *Options ➤ Next Form Language* or *Previous Form Language.*

Other languages maybe available but are not displayed due to the *Datamodel properties* option to restrict the available languages. *See Chapter 3, Section 3.8, Datamodel properties.*

6.10.6 Multimedia

The Blaise DEP commands for multimedia are accessed from the *Answer* menu and the *Options* menu. Select *Answer ➤ Play, Pause,* or *Stop* to play, pause, or stop the contents of the media language for that field. This might be a sound file, a video, or a series of images. If you have configured the DEP to automatically play multimedia files, the file will display or begin to play automatically when you reach the appropriate field.

To turn sound on and off, select *Options ➤ Mute.* To hide or show a video on the screen, select *Options ➤ Hide Video*. You can also use the DEP Menu Manager to program function keys to perform these options. For more information on using multimedia in data models, see Chapter 5.

6.10.7 Watch window

The Watch Window is a handy debugging tool that can be used to inspect values of fields and auxfields, to inspect which block have been checked and to inspect which external files have been accessed. The watch window is activated under *Run ➤ Parameters ➤ Dep* in the Control Centre or via the command line option /!.

Use the watch window pop-up menu to access to the following settings:

- *Stay on top*. When set the watch window will stay on top of the DEP and will always be visible.
- *Select fields*. Use this option to access the field selector. In the fields selector select the fields and auxfields to watch in the watch window.
- *Clear selected fields*. When choosing this option the current selection of (aux) fields will be cleared.
- *Show question text*. When set the current question text will be displayed for the selected (aux)fields.
- *Show block checks*. When set the system will display which blocks have been checked during the last execution of the rules. The time when the block check started will be displayed.
- *Show external file access*. When set the system will display the external files that have been accessed for search or read during the last execution of the rules. The key value used for the access will be displayed.
- *Load watch settings*. Use this option to load previously stored watch settings.
- *Store watch settings*. Use this option to store the current settings of the watch window. The current settings of the watch window are saved by default in the watch window settings file when the watch window is closed. The name of this file is the name of the data model file and it has extension `.bww`. When using the Watch Window command line option /!, behind the ! you can optionally specify the name of a watch window settings file.
- *Copy*. Use this option to copy the current contents of the watch window to the clipboard. The copy is cumulative: The current content is added (in front) to what previously has been copied to the clipboard.

6.11 Running the DEP Outside the Control Centre

During development you can run the DEP from the Control Centre. In production, you will almost always run the DEP from outside the Control Centre. There are several reasons for this. Users do not need to know anything about the Control Centre, since it would confuse them and give them options not meant for interviewers. And in most circumstances, an instrument will be run from a laptop or using the CATI Call Management System.

There are three main ways you can run the DEP from outside the Control Centre:

- You can create a Windows® shortcut to run the DEP and the appropriate instrument.
- You can use Maniplus to run the DEP. See the Maniplus guide for information.
- You can also create a batch file to run the DEP.

7 Basic Manipula

Manipula is a data processing system that can select, convert, rearrange, and sort data. It can combine data, derive new data, sort records, define filters, and perform complex computations.

Almost every study you conduct with Blaise® will require some use of Manipula. This is because there is usually a need to move and manipulate files of data.

Manipula is a fast and powerful tool with specially designed features to support efficient file processing, particularly of Blaise® files, but other types as well. Manipula can read data in four types of format and can write data in five types of format:

- Standard ASCII text files (positional or delimited formats)
- Special Blaise ASCIIRelational text files (positional or delimited formats)
- Special Blaise fixed text files (random access files with no end-of-record markers)
- Special Blaise print text files (formatted pages with headers and footers)
- Blaise data files created as part of interviewing or data editing (can only be read or written by Blaise)

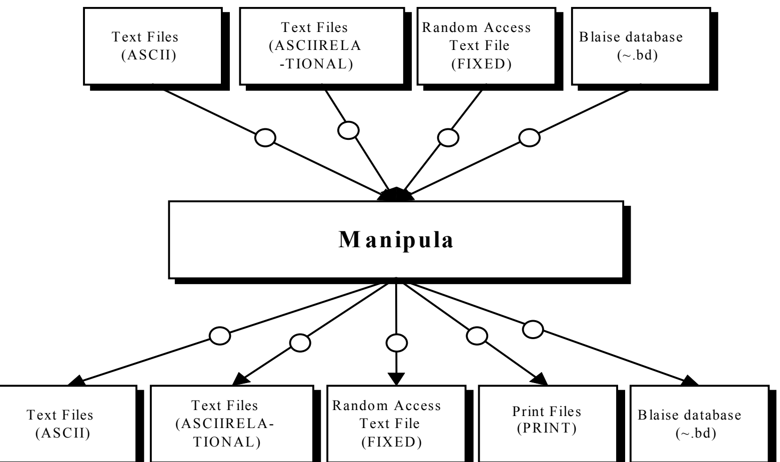
Beginning with version 4.5, Blaise includes advanced capabilities that allow Manipula to read and write directly to many PC databases, both client-server databases such as Oracle® and Microsoft® SQL Server™, and desktop databases such as Microsoft Access®. The capability is part of the Blaise Component Pack (BCP). The BCP is a separately licensed enhancement to the standard Blaise system, and is documented in a separate manual. Our description of Manipula in this manual will address only capabilities in the standard Blaise system.

In Manipula, there can be multiple input and output files, and each file might be of a different type of format. Each individual data file needs to have a description of the structure of records in the file so that the data can be read and written. This structure is defined in Manipula by using a data model description.

The following figure shows the types of data that can be read and written by Manipula. For each connection line in the figure, there must be a data model

description, represented by a circle in the figure. More complete explanations of these formats are in Section 7.5.

Figure 7-1: Flow of data for Manipula



This chapter covers basic Manipula programming where there is one input file, one or two output files, and the file manipulation features of Manipula are automatically invoked. More advanced uses of Manipula are covered in Chapter 8. An interactive enhancement to Manipula, called Maniplus, is covered in a separate manual.

7.1 Things You Can Do With Manipula

There are many things you can do with Manipula for both basic and sophisticated needs.

- If you have unique identification numbers and associated administrative data, you can initialise a Blaise data set before the survey starts. You can manipulate and reformat the data before initialisation.
- You can read survey data into or out of Blaise, or add data stored on laptops as part of a Computer Assisted Personal Interviewing (CAPI) study to the central database. You might want to do this for a section of the questionnaire or for a subset of forms.

- You can generate reports on survey progress.
- You can run an edit check on all forms or a subset of forms in a data file, invoking the RULES sections of the data model. The data may have come from another source, or you may need to apply additional edits or computations after data collection and before a final edit review.
- If you have already collected some data and find that you need to modify the data model, you can convert the existing data into the new data format. This can be done during development or production.
- You can export Blaise data to another software package such as SAS or Oracle, or write text files (such as batch files) based on the contents of a Blaise data model.
- You can derive new fields from the fields in your survey. Some examples are making a new income field by splitting the measured income into a number of classes, or making a new income field by summing all income components.
- You can sort an ASCII file on certain fields or calculate simple statistics on certain fields depending on the values of the sort fields.
- You can construct data records from information contained in several different input files into one or more output files. The record construction is based on some common link field or group of common link fields.

7.2 Starting Manipula

A Manipula setup is a text file that normally has the extension `.man`. If the extension differs, then Blaise does not automatically treat the file as a Manipula setup.

7.2.1 Creating a Manipula setup

There are two ways to create a Manipula setup from the Control Centre:

- Use the *Manipula Wizard* option from the *Tools* menu to create the setup based on the results of some simple questions.
- Create a setup in the Blaise data editor.

Using the Wizard is a simple way to get started. Once you are proficient in Manipula programming, you can create a setup in the text editor.

The following section describes how to create a setup using the Wizard and provides an example. For information on creating a setup in the text editor, read the following sections in this chapter.

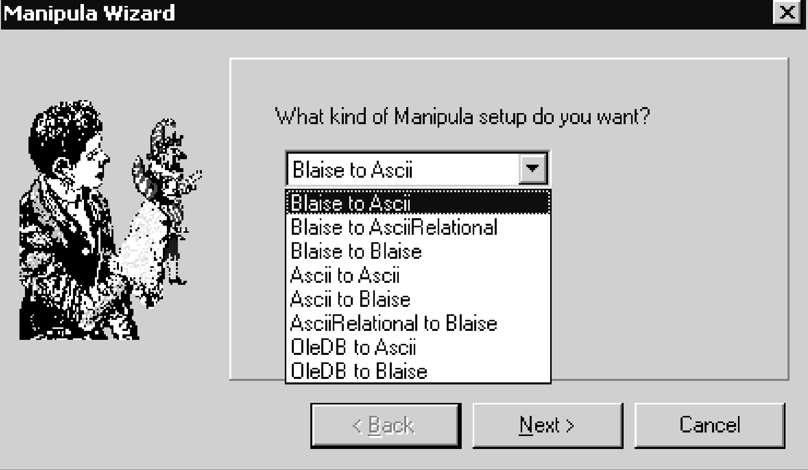
Creating a Manipula setup using the Manipula Wizard

As an example, suppose you have carried out a survey in which you asked questions relating to personal characteristics. This data model is the file `namejob1.bla` found in `\Doc\Chapter7`. ASCII data for the data model are in `namejob1.asc`. The task is to use the ASCII data to initialise a Blaise database file (`namejob1.bdb`).

The first step is to prepare `namejob1.bla` to produce the metadata file (`namejob1.bmi`).

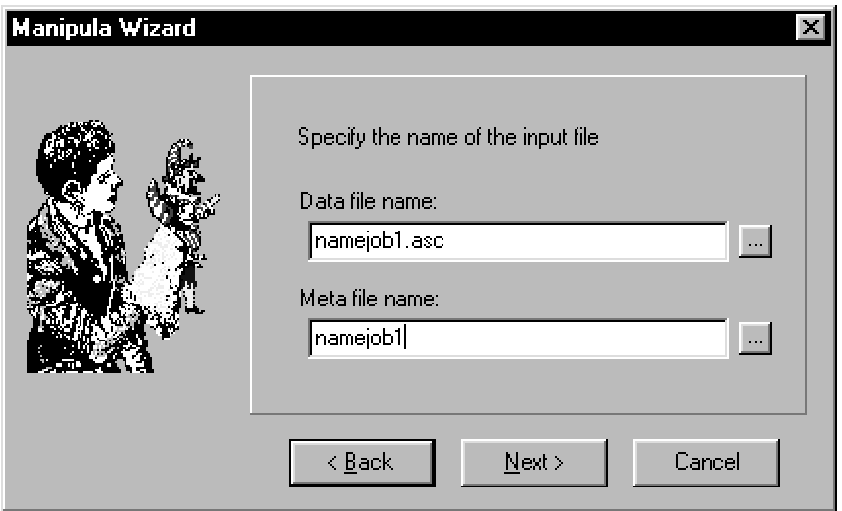
We can now create a Manipula setup using the Manipula Wizard. Select *Tools ➢ Manipula Wizard* from the menu, and the *Manipula Wizard* dialog box appears. In our example, we will select the *ASCII to Blaise* option.

Figure 7-2: Manipula Wizard



When you click the *Next* button, the following box appears:

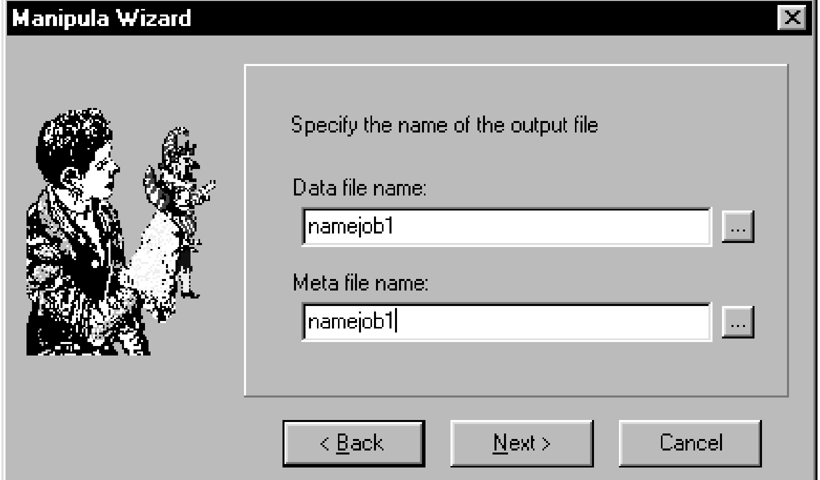
Figure 7-3: Naming the input file in the Manipula Wizard



In our example, data are read from a text file `namejob1.asc` whose description is given by the data model `namejob1` (saved in the file `namejob1.bmi`). The next window (not shown) asks if the data in the file are delimited and asks you to provide any separator or delimiter marks.

After the questions about whether the file is delimited, the following box asks for information about the Blaise file to which the data are written:

Figure 7-4: Naming the output file in the Manipula Wizard



In our example, data are written to a Blaise database file `namejob1.bdb` whose description is given by the data model `namejob1.bla`, as saved in the file `namejob1.bmi`.

In the final window (not shown), you specify the name you are going to give the Manipula setup file. We will use the name `frmascii.man`.

The frmascii.man example

The `frmascii.man` setup created by the Wizard is:

```
SETTINGS
  DESCRIPTION = 'ASCII to BLAISE'

USES
  InputMeta 'NameJob1'


INPUTFILE INPUTFILE1: InputMeta ('NameJob1.asc', ASCII)

OUTPUTFILE OUTPUTFILE1: InputMeta ('NameJob1', BLAISE)

MANIPULATE
  OUTPUTFILE1.WRITE
```

This setup will automatically copy every form from the input to the output data set without conditions, overwriting everything that was already there. The data definition of the output ASCII file can be found in the file `\Doc\Chapter7\namejob1.dic` (produced by the Cameleon translator `dic.cif)`. See Chapter 9 for information on Cameleon.

The `frmascii.man` setup could have been produced by the text editor in the Control Centre, and this is where more complex setups have to be created.

7.2.2 Preparing a Manipula setup

Once you have created the setup you need to check the program syntax. This is known as *preparing* the setup. Do this by using *one* of the following:

- Selecting *Project* ➤ *Prepare* from the menu.
- Pressing F9.
- Clicking the *Prepare* speed button on the Speedbar.

If the syntax is correct, a special executable file that has extension `.msu` is created.

7.2.3 Running a Manipula setup

Once you have prepared the setup, you execute or run the setup by opening the Manipula setup itself (the file with the `.man` extension). Then press Ctrl-F9, or click the *Run* speed button, or select *Run* ➢ *Run* from the menu.

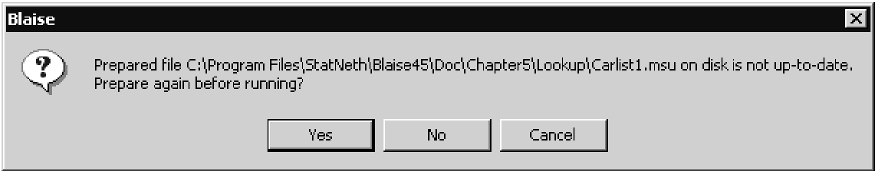
When you use *Run,* Blaise checks to see which file is in the current window: a Blaise data model, a Manipula setup, or a Cameleon translator. Blaise then checks to see if any Run parameters have been set for that type of file. If parameters have been set, then these replace any parameters specified in the file. Therefore, before you run the file, make sure the parameters have been set correctly or are left blank. Run parameters are discussed later in this section.

If there is a primary file set for the project, the primary file is run, not the file in the active window.

If the `.msu` file does not exist, it is automatically prepared. If the `.msu` file already exists, then the existing `.msu` file will be executed.

If you have changed the setup (`.man`) since the `.msu` file was prepared, there will be a warning message:

Figure 7-5: Sample warning message when running Manipula

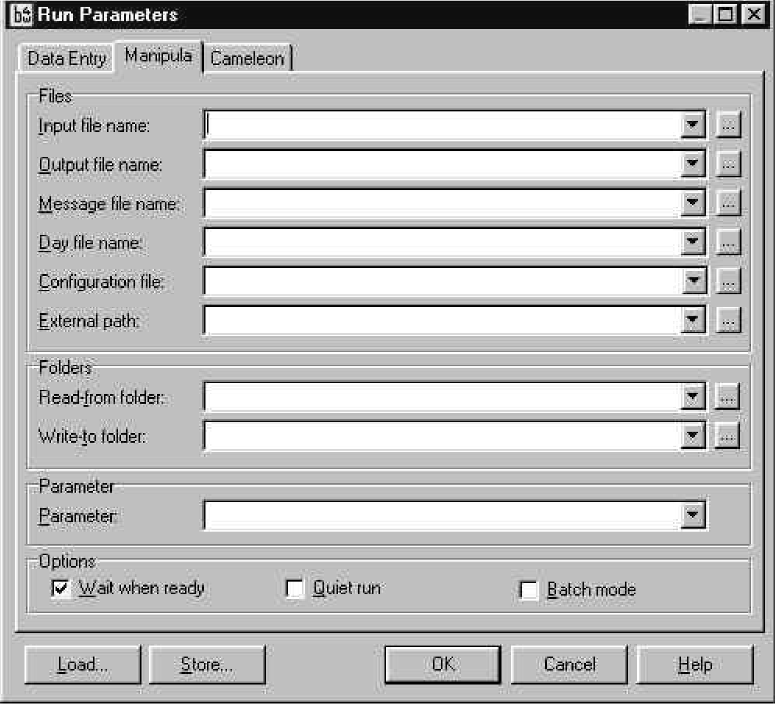


7.2.4 Manipula Run parameters

You can set run parameters for Manipula. These will affect all Manipula setups run from the Control Centre.

Select *Run* ➢ *Parameters* from the menu and the *Run Parameters* dialog box appears.

Figure 7-6: Run Parameters dialog box for Manipula



In all the entry boxes, only enter information if you want to change the defaults for the setup you will run. These parameters will remain in operation for all subsequent Manipula setups until changed.

- *Input file name.* Specify the name of the input file in the setup by typing the name, or by selecting from previous files by clicking on the down arrow. Multiple file names can be separated with a comma.
- *Output file name.* Specify the name of the output file in the setup by typing the name, or by selecting from previous data files by clicking on the down arrow. Multiple file names can be separated with a comma.
- *Message file name.* Specify the name of the message file (See *Section 7.8.4* for more information on message files and day files).
- *Day file name.* Specify the name of the day file, which is an optional log file created when the setup is run.
- *Read-from folder.* Specify the name of the folder from which you want to take the input.
- *ConfigurationFile.* Name of the configuration file (extension .MIW).

- *External path.* The path to search for external data files used during checkrules. The path only applies to external files with no absolute path specified.
- *Write-to folder.* Specify the name of the folder to which you want to direct the output.
- *Parameter.* Specify up to 32 parameters required by the setup. Parameters are accessed in a Manipula setup by the use of PARAMETER(1), PARAMETER(2), … in the MANIPULATE section of the setup.
- *Wait when ready.* Check to have Manipula prompt you when it is finished so that you can inspect the results of record counters on screen.
- *Quiet run.* Check to run Manipula in *quiet* mode. This runs Manipula without showing the Manipula dialog boxes on the screen.
- *Batch mode.* Check to instruct Manipula to run in batch mode. In this case, all user interaction will be suppressed (for instance the PAUSE instruction and WAIT parameter of a DISPLAY command will be ignored).

When finished, click the *OK* button.

7.3 Inspecting Input and Output Data

During development and testing, you can inspect output data in the Control Centre.

For ASCII input or output files, you can use a normal window in the text editor. For example, this is the original ASCII file (compare to the following Blaise data):

Figure 7-7: Viewing ASCII data for namejob1,
used as input data by frmascii.man

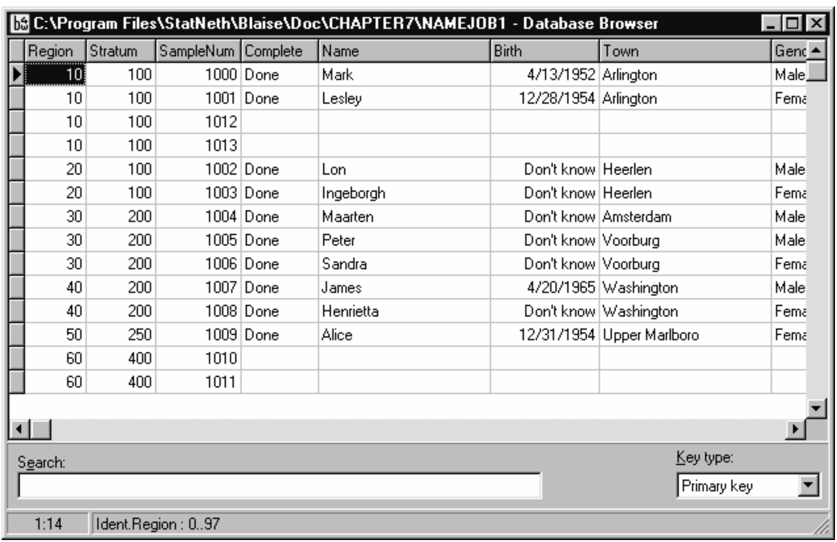
C:\Program Files\StatNeth\Blaise\Doc\CHAPTER7\Namejob1.asc

```
110 1001000Mark                    13041952Arlington           11
110 1001001Lesley                  28121954Arlington           22
 10 1001012
 10 1001013
120 1001002Lon                     99999999Heerlen             11
120 1001003Ingeborgh               99999999Heerlen             21
130 2001004Maarten                 99999999Amsterdam           11
130 2001005Peter                   99999999Voorburg            11
130 2001006Sandra                  99999999Voorburg            21
140 2001007James                   20041965Washington          12
140 2001008Henrietta               99999999Washington          21
150 2501009Alice                   31121954Upper Marlboro      22
 60 4001010
 60 4001011
```

1: 1 Insert

For Blaise output files, use the Database Browser. For example, for `namejob1.bdb`, you can see all or a subset of the data fields.

Figure 7-8: Blaise data for namejob1, created by frmascii.man



C:\Program Files\StatNeth\Blaise\Doc\CHAPTER7\NAMEJOB1 - Database Browser

Region	Stratum	SampleNum	Complete	Name	Birth	Town	Genc
10	100	1000	Done	Mark	4/13/1952	Arlington	Male
10	100	1001	Done	Lesley	12/28/1954	Arlington	Fema
10	100	1012					
10	100	1013					
20	100	1002	Done	Lon	Don't know	Heerlen	Male
20	100	1003	Done	Ingeborgh	Don't know	Heerlen	Fema
30	200	1004	Done	Maarten	Don't know	Amsterdam	Male
30	200	1005	Done	Peter	Don't know	Voorburg	Male
30	200	1006	Done	Sandra	Don't know	Voorburg	Fema
40	200	1007	Done	James	4/20/1965	Washington	Male
40	200	1008	Done	Henrietta	Don't know	Washington	Fema
50	250	1009	Done	Alice	12/31/1954	Upper Marlboro	Fema
60	400	1010					
60	400	1011					

Search:

Key type: Primary key

1:14 Ident.Region : 0..97

7.4 Basic Operation of Manipula

Tasks performed by Manipula

When Manipula is executed, several tasks are performed by the system, automatically if appropriate:

- Data files are opened and closed as necessary.
- A data tree, a representation of the data corresponding to the structure of a data model, is constructed in memory. This is done once for the input and output data models separately. If there are two or more data trees, a connecting scheme looks for identically named fields between data models.
- Data are read into memory, one record at a time, from the input data file. Before each record is read in, the contents of the last input and output record are deleted from memory. In memory, data are copied from the first data tree (belonging to the input file) to the second (belonging to the output file).
- If there is a MANIPULATE section (see *Section 7.6.4*), then data manipulations are carried out on the output data record while still in memory. After the data copy and manipulation, output data are written to the output file on disk and the process starts anew.

Default settings

There are several default settings that are applied to the Manipula setup unless turned off. These are designed to simplify the use of Manipula. For example, you do not have to bother about read and write statements. These automatic settings include the following:

```
AUTOREAD       Reads input file record.

AUTOCOPY       Copies data from the input file to the output file in
               memory for identically named fields.
```

An implicit feature of Manipula (not a setting) is:

```
AUTOWRITE
```

If a Manipula setup does not have a MANIPULATE section, data are automatically written to the output file. As soon as there is one instruction in the MANIPULATE section, then AUTOWRITE is turned off and you have to write the record yourself.

For more sophisticated uses of Manipula, some of the automatic settings are not appropriate. Manipula settings which you can use to customise the behaviour of Manipula setups are covered later in this chapter.

7.5 File Formats Supported by Manipula

Manipula supports six file formats for input and output data files. The first three file types are covered in this chapter.

- Manipula can read and write Blaise data files, provided a description of these files is available in the form of a Blaise metadata file in the USES section. This means that there is a Blaise data model prepared. To indicate this data file type, use the reserved word BLAISE.
- Manipula can read and write text files that can be produced by text editors, Manipula, database systems, statistical analysis systems, or programming languages. Text files can have separators/delimiters or use positional format, but all records must have end-of-record markers. Blaise accepts *Carriage-Return* (ASCII *13*), *Carriage-Return, Line-Feed* (ASCII *13-10),* and *Line-Feed (ASCII 10*) as end-of-record markers. (Text files that come from UNIX systems have ASCII *10* as end-of-record markers.) Records do not have to be the same length. At present, export or import of data for Blaise uses text files. Normally, if a text file is held on disk, only sequential read operations can be carried out. Random access is possible, however, if the file is stored in memory (as with Manipula temporary files). Using binary search techniques, with files stored in memory, records can be located quickly. To indicate text files, use the reserved word ASCII.
- Manipula can write a print file. A print file is a text file that you can use for reports. You can have headers, footers, page and line feeds, and formatted text that appear on every page or selected pages. If you want to use an output file as a print file, you can specify the file type PRINT in the OUTPUTFILE section. For details, see *file type* in the online reference manual.

The use of the next two file types is covered in Chapter 8.

- Manipula can read and write ASCIIRelational files. These files are used for importing and exporting from a relational database to a Blaise data set, or vice versa. ASCIIRelational causes a data set to be written for each block definition unless the block is embedded within a higher level block. Regardless of the direction of the flow, data are written to ASCII as an intermediate step.

- Manipula can read and write a special kind of ASCII file called a *fixed* file (or random access file). This is an ASCII file that does not have end-of-record markers. All records must be the same length. By using binary search techniques, records can be located quickly. Searching can be done either on disk, if the file is sorted on the values of a key field, or in memory. Such files are indicated with the reserved word FIXED.

The sixth file format, OLEDB, provides access to relational databases such as Microsoft Access®, Microsoft® SQL Server™, and others. It does this using Microsoft's database technology-- ADO/OLEDB. This advanced capability is not part of the standard version of Blaise but comes with the added-cost Blaise Component Pack (BCP) product. OLEDB in Blaise is documented separately.

7.6 Outline of a Basic Manipula Setup

Manipula setups consist of a number of sections. The most important sections are:

- USES
- INPUTFILE
- OUTPUTFILE
- MANIPULATE

In addition, you can include the following sections:

- UPDATEFILE
- TEMPORARYFILE
- SORT
- PRINT
- AUXFIELDS
- SETTINGS

We will illustrate each section through examples and discuss each section briefly. You will find more details in the *Reference Manual*.

7.6.1 USES section

The USES section declares the data models describing the data files used in the setup (a data model description). The data descriptions give the data definition and the structure of the data files. The section starts with the reserved word USES. Data model descriptions come in two forms.

```
USES
  MetaName 'MetaFileName'
```

MetaName is an identifier that refers to the Blaise data model describing the data files used in the Manipula setup. *MetaFileName* is the name of the Blaise metadata file. You can refer to *MetaName* in the INPUTFILE, UPDATEFILE, TEMPORARYFILE, or OUTPUTFILE sections, and you can include a path. For example:

```
USES
  BlaiseMeta 'C:\BLFILES\NAMEJOB1'
```

This specification assumes that there is a Blaise metadata file with the name `namejob1.bmi`. In other words, it assumes that you have prepared the data model `namejob1`. The meta name *BlaiseMeta* is used later in the Manipula setup to refer to the Blaise data model that describes the structure of a file. One or more input and output files can refer to one meta name.

```
USES
  DATAMODEL MetaName
    DataModelSpecification
  ENDMODEL
```

You can define a data model directly in the Manipula setup. Usually you use this for ASCII files for which Blaise data models do not exist and for which the description is not needed elsewhere. You may not use this explicit model definition for Blaise data files.

Again, *MetaName* is the identifier that refers to the data model in the Manipula setup. *DataModelSpecification* is the description of the structure of the data file. It can consist of FIELD sections, TYPE sections, and block definitions. The following is simple example (`asc2asc.man`):

```
USES
  DATAMODEL FlatFile1
    FIELDS
      Completion : (Done, NotDone)
      Region     : INTEGER[2]
      Stratum    : INTEGER[4]
      SampleNum  : 1000..9000
      Name       : STRING[20]
      Birth      : DATETYPE
      Town       : STRING[20]
      Gender     : (Male, Female)
      Job        : (Yes, No)
  ENDMODEL
```

FlatFile1 is the identifier (meta name) that refers to the data model. The fields are defined after the reserved word FIELDS. Note that the field definitions have the same syntax as Blaise field definitions. Data models can be much more complex than the one above. You can define blocks and arrays of blocks. You can have two or more data models in the USES section (see `split2.man`).

7.6.2 INPUTFILE section

The INPUTFILE section describes the names, data model references, and file types of input data files. The section starts with the reserved word INPUTFILE, and is terminated by the start of a subsequent section:

```
INPUTFILE
  DataName: MetaName ('DataFileName', FileType)
```

- *DataName* is an identifier used to refer to this input file in later sections of this setup.
- *MetaName* is the identifier referring to the data model as specified in the USES section.
- *DataFileName* is the name of the file containing the data (specified within quotes). The file name may include a path specifying the folder containing the file, and a file extension is allowed (except for Blaise and ASCIIRelational files).
- *FileType* denotes the type of the data file. You can write one of the reserved words BLAISE, ASCII, FIXED, or ASCIIRELATIONAL. If you omit the file type, the default type ASCII is assumed.

Suppose we want to read an ASCII file corresponding to the data model description (*FlatFile1*) in the USES section above. The following is one possibility for the INPUTFILE section (`asc2asc.man`):

```
INPUTFILE
  InFile : FlatFile1('NameJob1.asc', ASCII)
```

The meta name of the data model is *FlatFile1*. This identifier, already defined in the USES section, is used in the INPUTFILE section to tell Manipula the structure of the text file `namejob1.asc`. We could have left out the file type, since ASCII is the default file type.

You can describe more than one input data file. In that case, the first input file is called the main input file, and all other input files are called link files. You need a separate INPUTFILE section for each input file. This means you need the key word INPUTFILE for each input file. Use of multiple input files, including linking them, is covered in Chapter 8.

7.6.3 OUTPUTFILE section

Specifying output files is very similar to specifying input files. You use the reserved word OUTPUTFILE. For identically named fields (taking into account dot notation) that are mentioned in both the input and output files, the values are automatically copied from input to output. Note that if the output field is not type compatible with the identically named input field, Manipula will report an error during the parsing stage of the setup. Fields that only appear in the output file initially remain empty. However, for every output field a new value can be computed in the MANIPULATE section.

You can specify more than one output file. In that case, you have to include an OUTPUTFILE section for each file. This means you must have the key word OUTPUTFILE for each output file. This allows you to split files by writing some records to one file and writing different records to another file. For example:

```
OUTPUTFILE
  CompleteData : BlaiseMeta('Complete.asc', ASCII)

OUTPUTFILE
  MissingData : BlaiseMeta('Missing.asc', ASCII)
```

An example illustrating one file split into two is `split1.man`.

7.6.4 MANIPULATE section

In the MANIPULATE section, you specify computations and derivations of fields. Using conditional statements such as IF … ENDIF, you can subset data files, or

process selected fields. You can define procedures, then reuse them with different fields. The section starts with the reserved word MANIPULATE.

For some Manipula setups, the MANIPULATE section is not required. Without it, records will automatically be written to an output file. However, once the MANIPULATE section contains any instruction, you must use the WRITE instruction to write records to an output file.

Expressions

You can assign new values to output fields and auxfields by using expressions. The results of these expressions are assigned to output fields. Auxfields are discussed in *Section 7.8.1*. Expressions are formally discussed in Chapter 3. The use of expressions is illustrated in a number of examples given below and in the folder `\Doc\Chapter7`.

Functions

There are many functions that you can use in the Manipula setup. Functions are explained and listed in the *Reference Manual*. The Manipula setup `prog2.man` illustrates the use of a string function STR.

Control structures

There are a number of control statements that help you control the logic of the manipulations. Manipula offers a few more control structures than Blaise. Besides the IF-statement and the FOR-loop, Manipula offers the CASE-statement, the WHILE-loop, and the REPEAT-loop. The IF-statement and FOR-loop have already been discussed in Chapter 3. The following is an example of an IF condition in a MANIPULATE section:

```
MANIPULATE
  IF (Working = 1) AND (Distance > 20) THEN
    Commuter := 1
  ELSE
    Commuter := 0
  ENDIF
  OutFile.WRITE
```

This is an efficient structure since the assignment to the field *Commuter* is only executed once. Since there is a MANIPULATE section with at least one instruction in it, the WRITE instruction is needed to write records to the output data set.

CASE OF statement versus IF-ELSIF-THEN statement

When it comes to complex, excluding conditions, you can use the CASE OF structure in Manipula. The first example below uses an IF-ELSEIF-THEN structure to determine age classes in the field *AgeClass*.

```
IF Age < 20 THEN AgeClass:= 1
  ELSEIF Age < 40 THEN AgeClass:=2
  ELSEIF Age < 60 THEN AgeClass:=3
  ELSE AgeClass:=4
ENDIF
```

This second example accomplishes the same thing using the CASE OF structure.

```
CASE Age OF
   1..19:  AgeClass:= 1
   21..39: AgeClass:= 2
   41..59: AgeClass:= 3
  ELSE AgeClass:=4
ENDCASE
```

While the results of the two structures are identical, the CASE OF structure is easier to understand.

Two MANIPULATE sections

You can have two MANIPULATE sections in one Manipula setup as long as they are separated by a SORT section.

7.6.5 Other file sections

Other file sections include UPDATEFILE and TEMPORARYFILE. These sections are covered in Chapter 8.

7.7 Basic Examples

The following examples use only basic features of Manipula: the USES, INPUTFILE, OUTPUTFILE, and MANIPULATE sections. The relevant data model, Manipula setups, and data files shown below are found in `\Doc\Chapter7`.

7.7.1 Extending a Blaise data file

In development or in production, you might find it necessary to modify your Blaise data model. In certain situations, this will change the data definition of the data set (see Chapter 3). If you have already collected data, you will not want to re-enter them, but you would like to have all of the data held in the same Blaise data file.

In this situation, you can read data directly from the old version of the data model to the new version. The example that follows is used in development situations, but the principle is the same for data models that are already in production. Here is the Manipula setup `oldtonew.man`:

```
USES
  OldDataModel  'OLD\NameJob1'
  NewDataModel  'NameJob1'

INPUTFILE
  InFile : OldDataModel ('OLD\NameJob1', BLAISE)

OUTPUTFILE
  OutFile : NewDataModel ('NameJob1', BLAISE)
```

In this example, there is a folder named `OLD` under the current work folder. The subfolder contains the `namejob1.b*` files for the `namejob1` data model. To create a new version of the data model, copy all files of the form `namejob1.b*` to the `OLD` folder, and then delete the `namejob1.b*` files from the current work folder.

After modifying the data model by changing the `.bla` file in the work folder and then preparing it, run the Manipula setup above. This will populate the new database with the data previously entered. Manipula will transfer all data with matching names, taking into account block names, if appropriate, that are type compatible.

7.7.2 Initialising a Blaise data file

You can initialise a Blaise data file through an ASCII to Blaise import. The ASCII file will hold only administrative data such as unique identification numbers and names. This is useful for Computer Assisted Telephone Interviewing (CATI) surveys, when you need to start with telephone numbers to dial.

For example, the ASCII data set may might look like the following (`init.asc`):

```
 10 1001000Mark
 10 1001001Lesley
 10 1001012Esme
 10 1001013Vita
 20 1001002Lon
 20 1001003Ingeborgh
 20 1001004Denise
```

The Manipula setup that will read in this data file is called `init.man` and is:

```
USES
  BlaiseMeta 'NAMEJOB1'

INPUTFILE
  InFile : BlaiseMeta('INIT.ASC', ASCII)

OUTPUTFILE
  OutFile : BlaiseMeta('NameJob1', BLAISE)
```

If all of the records of the starting ASCII file are to be read in, then the Manipula setup above will suffice. If you want to read in only those records with known names, you need the following MANIPULATE section:

```
MANIPULATE
  IF NAME <> '' THEN
    OutFile.WRITE
  ENDIF
```

7.7.3 Exporting a Blaise data file to ASCII

To export all data from a Blaise data file to ASCII, use a setup similar to the following (`toascii.man`):

```
USES
  BlaiseMeta 'NAMEJOB1'

INPUTFILE
  BlaiseData: BlaiseMeta('NameJob1', BLAISE)

OUTPUTFILE
  AsciiData: BlaiseMeta('NameJob1.asc', ASCII)
```

This Manipula setup was written by the Cameleon translator `toascii.cif`, and will write all data records and all fields to the ASCII data file. To get a description of the data file, run the Cameleon dictionary setup `dic.cif`. This setup is sufficient for simple data models. If you have a hierarchical data model,

then you will want more sophisticated ways of writing data, which are discussed in Chapter 8.

To write only the records which are complete, use a Manipula setup with a MANIPULATE section. For example (`toascii2.man`):

```
USES
  BlaiseMeta 'NAMEJOB1'

INPUTFILE
  BlaiseData : BlaiseMeta('NameJob1', BLAISE)

OUTPUTFILE
  AsciiData : BlaiseMeta('NameJob2.asc', ASCII)

MANIPULATE
  IF BlaiseData.Complete = Done THEN
    AsciiData.WRITE
  ENDIF
```

If you want to split the output into two files, one file for complete forms and the other file for incomplete forms, you can use two OUTPUTFILE sections and appropriate instructions in the MANIPULATE section. For example (`split1.man`):

```
USES
  BlaiseMeta 'NAMEJOB1'

INPUTFILE
  BlaiseData : BlaiseMeta('NameJob1', BLAISE)

OUTPUTFILE
  CompleteData : BlaiseMeta('Complete.asc', ASCII)

OUTPUTFILE
  MissingData : BlaiseMeta('Missing.asc', ASCII)

MANIPULATE
  IF BlaiseData.Complete = Done THEN
    CompleteData.WRITE
  ELSE
    MissingData.WRITE
  ENDIF
```

7.8 Extending a Manipula Setup

Four additional sections that are commonly used in Manipula are AUXFIELDS, SORT, PRINT, and SETTINGS.

7.8.1 AUXFIELDS section

Auxfields in Manipula have much the same use as they do in a Blaise data model. They hold intermediate values in calculations that can be used later in the setup. The AUXFIELDS section is defined after the last file section, and appears before the MANIPULATE section. The syntax for auxfields is the same as that for fields.

Auxfields are initialised every time a record is read from the first input file. They do not carry over values from one record to another.

Sometimes an auxfield is used to hold an intermediate value is in `prog1.man`, part of which is shown in the following example:

```
AUXFIELDS
  CompletionString : STRING[15]

MANIPULATE
  IF BlaiseData.Complete = Done THEN
    CompletionString := 'Complete'
  ELSE
    CompletionString := 'Not Complete'
  ENDIF
```

GLOBAL auxfields

Sometimes it is necessary to hold the value of a field from one input record to the next. For example, an ASCII input file may be sorted on the completion status field. In this case, all of the complete forms are together and all of the incomplete ones are together. You might want to detect when you change from the part of the data set with complete forms to the part where there are incomplete forms. You can do this with a global auxfield. They are not initialised for every record as normal auxfields are. Global auxfields are declared in a separate AUXFIELDS section. The syntax is:

```
AUXFIELDS (GLOBAL)
```

The following is an example from `prog2.man`:

```
AUXFIELDS (GLOBAL)
  HoldCompletionString : STRING[15]

{Many lines and sections later.}

MANIPULATE
  IF HoldCompletionString = '' THEN
    ProgressReport.PRINTSTRING('Completed Reports')
    HoldCompletionString := CompletionString
  ELSEIF CompletionString <> HoldCompletionString THEN
    ProgressReport.PAGE
    ProgressReport.PRINTSTRING('Missing Reports')
    HoldCompletionString := CompletionString
  ENDIF
```

In the above example, the value of *HoldCompletionString* will only change when an assignment is carried out. It will not be reinitialised when a new record is read.

A more complex use of global auxfields is found in Chapter 8, where you need to convert an ASCII file of several physical records per logical record to an ASCII or Blaise file of one physical record per logical record.

FOR-DO loops

Manipula does not allow locals as Blaise does. When you use a FOR-DO loop in Manipula, use an auxfield as the counter. See Chapter 8 for an illustration of a FOR-DO loop.

7.8.2 SORT section

When you have a text output file, it is often necessary to sort it on certain fields before continuing processing. For example, you might have a file with complete and incomplete forms mixed together. When the data are read out to ASCII, you can sort the output records on the field for completion status. Then you can process the result further. A SORT section is found in the setup `prog2.man`.

```
SORT
  CompletionString  (ASCENDING)
  OneLine           (ASCENDING)
```

Here the output ASCII file is sorted in ascending order on the two fields *CompletionString* and *OneLine.*

Reserved words that can be used with a sort include ASCENDING, DESCENDING, SUM, MEAN, MAXIMUM, MINIMUM, STDDEV, VARIANCE, and MEDIAN. Enclose these words in parentheses after the name of the field to which they apply.

The reserved words SUM, MEAN, MAXIMUM, MINIMUM, STDDEV, VARIANCE, and MEDIAN indicate a special treatment for all records of which the sort fields have the same value. They can be used for numeric fields.

```
SORT
  Age    (ASCENDING)
  Gender (DESCENDING)
  Income (SUM)
```

For each different value of the sort fields, one record will be present after the sort. For example, the field Income in each record will have the sum of the Income field of all records with the same value of the sort fields.

A SORT section is often followed by another MANIPULATE or PRINT section. You cannot sort Blaise files (but you do not need to sort Blaise files because you can process them in order of primary or secondary keys).

7.8.3 PRINT section

A PRINT section is used in conjunction with a print file declared in the OUTPUTFILE section. A print file is a text file with headers, footers, page or line breaks, and formatted text to make a report more readable. The following example is of a print file in an OUTPUTFILE section and a PRINT section found in the Manipula setup `prog2.man`:

```
OUTPUTFILE
  ProgressReport : Progress('prog2.asc', PRINT)

{Many lines and sections later.}

PRINT(ProgressReport)
  SETTINGS
    PAGELENGTH = 25
  HEADER := 'Progress report for survey'
  HEADER := 'Date : ' + DATETOSTR(SYSDATE) + '    ' +
            'Time : ' + TIMETOSTR(SYSTIME)
  FOOTER := 'Page ' + PAGENUMBER
```

The reserved word PRINT is followed in parentheses by the data name (in this case, *ProgressReport*) that has already been declared in the OUTPUTFILE section. Several reserved words can be used in conjunction with PRINT: SETTINGS, HEADER, FOOTER, and END. For more information on these key words, see the *Reference Manual*.

Date and time stamps

The reserved words SETTINGS, HEADER, FOOTER, and END produce text strings. Note the use of several functions in defining time and date stamps. The reserved words in the example above produce output files with headings such as:

```
Progress report for survey
Date : 01-04-1996   Time : 12:02:16
```

and a footer such as:

```
Page 1
```

You can also make date and time stamps in other kinds of files.

7.8.4 SETTINGS section

You can modify the behaviour or the output of a Manipula setup through the SETTINGS section. There are various ways to classify settings:

- They can be global or local.
- They can be file-related or not file-related.

Global settings

A SETTINGS section that is at the beginning of the Manipula setup is a global SETTINGS section. Settings made here affect all subsequent sections in the Manipula setup where applicable. An example of a global setting in `prog2.man` is:

```
SETTINGS
  DATEFORMAT = MMDDYY
```

This setting would be useful in the United States because it sets the date format to the convention used there. In any of the subsequent output files where there is a date stamp, the month will proceed the day of the month. The global setting above is the first section of the Manipula setup; this is what makes it global.

Figure 7-9 lists global settings that are not file-related. In this table, the name of the setting is in the first column, the syntax is in bold in the second column, and possible values for each setting are in the third column, with the default underlined:

! Some global settings can be set or overridden using command line parameters (see *Section 7.9* on batch processing using Manipula). Refer to the *Reference Manual* for complete details.

Figure 7-9: Global Manipula settings (not file related)

Setting	Description	Possible Values
AUTOREAD	Whether records are read automatically from the first (main) input file. **AUTOREAD = VALUE**	YES NO
CALCERROR	How to handle a run-time calculation error. **CALCERROR = VALUE**	CONTINUE MESSAGE HALT
DATEFORMAT	Format of a date string. **DATEFORMAT = VALUE**	DDMMYY MMDDYY YYMMDD
DATESEPARATOR	Separator in a string representation of a date. **DATESEPARATOR = 'S'**	Default is - Other possible values of S are ; : . , / \ # or a blank space.
DAYFILE	The name of a day file, an optional log file, created when running a setup. **DAYFILE = 'FileName'**	Default is no file. Setting can also be given on the command line.
DESCRIPTION	A setup text which is displayed on the screen and is written to the day file. **DESCRIPTION = STRING**	No description by default.
ESCAPE	Whether the current job can be interrupted. **ESCAPE = VALUE**	YES NO
INPUTPATH	The default folder for input files that do not include a DOS path. **INPUTPATH = 'PathName'**	Default is no input path. Can also be given on the command line.
MAXMESSAGE	The maximum number of messages to be written to the message file. If this number is reached, Manipula will halt execution. **MAXMESSAGE = N**	N = 100. If you set N = 0, there is no maximum.

Setting	Description	Possible Values
MESSAGEFILE	The name of the message file. **MESSAGEFILE = 'FileName'**	Manipula.msg Can also be given on the command line.
METASEARCHPATH	The default folder for the meta information files. **METASEARCHPATH = 'PathName'**	Default is no search path. Can also be set from the command line.
OUTPUTPATH	The default folder for output files that do not include a DOS path. **OUTPUTPATH = 'PathName'**	Default is current folder. Can also be set on the command line.
OVERFLOW	How to treat a run-time string overflow error. **OVERFLOW = VALUE**	CONTINUE MESSAGE HALT
RANGEERROR	How to treat a range error. **RANGEERROR = VALUE**	CONTINUE MESSAGE HALT
SUBSCRIPTERROR	How to handle a run-time subscript error (for arrays). **SUBSCRIPTERROR = VALUE**	CONTINUE MESSAGE HALT
TIMEFORMAT	Specifies the format of a time in a string. **TIMEFORMAT = VALUE**	HHMMSS HHMM
TIMESEPARATOR	The separator in a string representation of a time. **TIMESEPARATOR = 'S'**	Default is: Other possible values are ; . / \ - , or blank space.
WARNINGS	Whether the list of run-time errors must be displayed on screen after execution. **WARNINGS = VALUE**	YES NO

Some organisations have standard settings that they always include in their Manipula setups.

For example:

```
SETTINGS
  DATEFORMAT = MMDDYY
  DATESEPARATOR = '/'
  DAYFILE = 'DAYFILE.LOG'
  DESCRIPTION = 'One Blaise data set into two ASCII.'
  MAXMESSAGE = 0
  MESSAGEFILE = 'MESSAGE.LOG'
```

These settings give a United States date format with '/' as a separator. The setup will continue to run no matter how many messages are written to the message file, since MAXMESSAGE = 0. A log file of Manipula steps will be kept in a file called `dayfile.log`, and messages, if any, will be written to `message.log`.

Note that the setting can be overridden on the command line as noted before. They can also be omitted entirely from the SETTINGS section above and still implemented using the command line, or some can be set by using the Run parameters. These settings are implemented in the setup `split2.man`.

File-related settings

File-related settings apply to INPUTFILE or OUTPUTFILE sections (also UPDATEFILE and TEMPORARYFILE sections discussed in Chapter 8). Some can be global or local, while others are more restricted. A local setting applies only to the INPUTFILE or OUTPUTFILE section in which it is declared. For example:

```
OUTPUTFILE
  ProgressReport : Progress('prog2.asc', PRINT)
  SETTINGS
    TRAILINGSPACES=NO
```

The local setting TRAILINGSPACES will affect only this output file, not any other output files that may be present.

Figure 7-10 on the pages that follow provides file-related settings. Check the *Reference Manual* for default settings as well as other details.

- In the Global/Local column, G = Global and L = Local settings.
- In the File Type column, A = ASCII, R = ASCIIRelational, P = Print, F = Fixed, B = Blaise, and blank = all file types.
- In the I/O column, I = Input files, O = Output files, U = Update, T = Temporary, and All = All input/output types.

Figure 7-10: File-related Manipula settings (local and global)

Setting	Global/ Local	File Type	I/O	Description
ACCESS	G/L	B	IOU	Whether the data file is for exclusive or shared use in Manipula.
AUTOCOPY	G/L		ITU	Copy input fields to output fields with matching names automatically.
CHARACTERSET **{ OEM | ANSI }**	G/L	B	I/O	With this setting you can specify the character set used. When used for an output file all data written to that file will be according to the ANSI character set. When used for an input file the system will assume all data will be according to the ANSI character set.
CHECKRULES	G/L		All	Check the rules of the data model.
CHECKRULES UNCHANGED	G/L		All	Whether the checkrules must be forced on an unchanged form.
CLEARSUPPRESSES ONCHECK	G/L		All	Whether the checkrules must reset all suppressed signals.
CONNECT	G/L		All	Prepare the data tree for copying of data to fields with equal names in other data trees.
DECIMALSYMBOL	G/L		I/O	With this setting you can specify the decimal symbol used in real type fields. The default setting is a period ('.').
DELIMITER	L	AR	IO	String field delimiter.
DUPLICATES	L	AP	O	Whether records with the same sort key may be present after the sort.
DYNAMICROUTING	G/L		All	Whether the checkrules must be carried out in dynamic routing mode.
EXCLUDEBLOCKS	L	R	O	Which block types have to be excluded from the ASCIIRelational export.
FILLZERO	G/L	APR	O	Whether numeric output fields must have leading zeroes.
FORMAT	L	AR	O	Format of fields in case a separator is present.
INCLUDEBLOCKS	L	R	O	Which block types have to be included in the ASCIIRelational export.
INITRECORD	L		OUT	Initialise the output record.

Setting	Global/ Local	File Type	I/O	Description
INMEMORY	L		I	Whether to load the entire input file in internal memory.
KEY	L	B	ITU	The key type by which the forms must be processed.
MAKENEWFILE	L		OU	Whether to append records to the current output file or to start a new one after deleting the old one.
ONLOCK	G/L	B	OU	Whether Manipula has to wait if a form is locked.
OPEN	G/L		IOU	Whether the file has to be opened when the setup is started.
RANGECHECK	G/L	AFPR	All	For field assignments, whether to check if the assigned value is within the valid range.
REMOVE	L	B	OUT	Whether the original record must be removed if the same record is saved under a different key.
REMOVEEMPTY	G/L		OU	Delete empty output files.
RENEW	L		OUT	Whether a newly produced record must overwrite an original record with the same key.
REQUESTED	G/L		IU	Whether a file must be present when the execution of a setup starts.
SELECTSTATUS	L	B	IU	Select records based on the form status of Clean, Dirty, Suspect, or NotChecked.
SEPARATOR	L	AR	IO	Field separator.
SEQUENTIAL	L	F	I	Whether to use a sequential or binary search method to link a file on disk.
SKIPEOF	L	A	I	Enable to continue reading records when EOF characters (ASCII #26) are encountered.
STARTKEY	L	B	I	Specify where to start processing the file based on the value of the key.
TRAILINGSPACES	G/L	APR	O	Whether trailing spaces of an output record must be written to the output file.

Which file-related settings you use depend on the application at hand. The best way to become familiar with these settings is to read the *Reference Manual*, to experiment, and to inspect examples of Manipula programs.

7.9 Running Manipula as a Separate Program

You can also run Manipula as a separate program. The syntax is:

```
MANIPULA SetupName Option Option ... Option
```

SetupName is the name of the prepared Manipula setup. You always have to specify this file. Manipula will always use the extension `.msu`. Therefore, you must first prepare your Manipula setup in the Control Centre before you can run it as a separate program outside the Control Centre.

Command line parameters for Manipula are in Appendix A. An example of a Manipula command line parameter is:

```
Manipula FRMASCII /IInit.asc
```

This command line of Manipula will use `frmascii.msu` but will override the input file specified in the original `frmascii.man`. Instead of reading in data from `namejob1.asc`, it will read in data from `init.asc`. Note that `init.asc` must be described by the same data model as `namejob1.asc`.

Another approach is to specify the Manipula options or parameters in a Blaise Command Line File (.BCF) and run Manipula with the BCF. For example:

```
Manipula @FRMASCII.BCF
```

FRMASCII.BCF is a text file like this:

```
[ManipulaCmd]
Setup=FRMASCII.MSU
InputFile=Init.Asc
```

For Manipula jobs using more than a few of the 16 available options, Blaise command line files can be quite helpful. They are easier to debug and can be generated for other processes more efficiently. The Blaise Command Line File capability is documented in Appendix A. Also, a list of all BCF options for both DEP.EXE and MANIPULA.EXE is given in Blaise.BCF in the Blaise installation root folder.

7.10 Example Manipula Setups

The following table lists the example setups used in this chapter. They can be found in the folder `\Doc\Chapter7`.

Figure 7-11: Example Manipula setups (listed alphabetically)

Manipula Setup	Description
asc2asc.man	From ASCII to ASCII with different data models.
frmascii.man	Survey data from ASCII to Blaise.
init.man	Administrative data from ASCII to Blaise.
oldtonew.man	From Blaise to Blaise for extending a Blaise data model.
prog1.man	Simple progress report.
prog2.man	More sophisticated progress report.
split1.man	From Blaise to two ASCII data sets.
split2.man	Shows global settings, complicated IF structures, three data models in the USES section, INCLUDE files, and block-level computations.
toascii.man	Survey data from Blaise to ASCII.
toascii2.man	From Blaise to ASCII for selected records.

8 Advanced Manipula

This chapter covers additional aspects of Manipula programming that were not covered in Chapter 7. This chapter explains how to run Manipula when there are multiple input files, multiple output files, and the file manipulation is under the user's control (that is, file manipulation is not automatically provided by Manipula).

All the sample files used in this chapter can be found in `\Doc\Chapter8` in the Blaise® system folder.

8.1 More Sections in Manipula

This section discusses additional sections you can include in a Manipula setup.

8.1.1 PROLOGUE section

The PROLOGUE section is like a MANIPULATE section, but it is invoked only once at the start of the setup. You can use it to initialise values of fields or auxfields before the MANIPULATE section is invoked.

```
AUXFIELDS (GLOBAL)
  GlobalCounter : INTEGER
PROLOGUE
  GlobalCounter:= 1000

MANIPULATE
  REPEAT
    InFile.READNEXT
    GlobalCounter:= GlobalCounter + 1
{etc}
```

The global auxfield *GlobalCounter* will start with value *1000* before the MANIPULATE section is invoked. The Manipula setups `maketest.man` and `many2one.man` use a PROLOGUE section.

8.1.2 UPDATEFILE section

The UPDATEFILE section allows you to change data in an existing Blaise® file. The update file declared in the UPDATEFILE section is available for both input and output operations. The syntax is similar to that of the INPUTFILE or OUTPUTFILE section. You can mix input and update sections in the same Manipula setup. For example:

```
UPDATEFILE
  UpFile : NCS07 ('NCS07', BLAISE)

INPUTFILE
  InFile : InitHH ('InitHH.asc', ASCII)
LINKFIELDS
  Region = UpFile.Ident.Region
  Stratum = UpFile.Ident.Stratum
  SampleNum = UpFile.Ident.SampleNum
```

This example is taken from `uphh.man`, a setup that updates existing Blaise forms with corrected information from an ASCII file.

With UPDATEFILE, you can read data into a Blaise data set in two or more stages, perform calculations on an entire data file, check the data against the rules in the RULES section of the data model, or update a data set with information from another file through links.

Caution with computations

Be careful with your assumptions about the values of the fields when you make computations. Once a field has been changed to a different value, subsequent computations, IF conditions, and the like will be executed with that changed value, not the original value of that field. There will be no record of the original value anywhere once the Manipula job is finished. Compare this to the use of INPUTFILE and OUTPUTFILE with and without data sharing in Section 8.2.1.

8.1.3 TEMPORARYFILE section

If your Manipula setup uses a temporary file (also called a scratch file) to hold intermediate values, then you can use a TEMPORARYFILE section. A temporary file is held completely in memory and is not stored on disk. For a large temporary file, you have to be sure you have enough memory available.

A temporary file is not saved after the setup executes, so you must ensure that you have transferred any data from the temporary file. Refer to the *Reference Manual* or the on-line help for details.

8.2 More About Files

In this section we discuss how to link files, the day file, a message file, customised information files, and the file methods WRITE, KEEP, WRITEALL, and KEEPALL.

8.2.1 Linking files and the LINKFIELDS subsection

When you have two or more INPUTFILE or UPDATEFILE sections, you usually want to link records between the files. When you link files, you associate a record in the main file (the first file) with a record in a link file (second or further file).

The link can be either static or dynamic. With a static link, Manipula finds matching records based on fields named in a LINKFIELDS section. Dynamic linking means that you have to control matching of records in the MANIPULATE section. A static link requires a LINKFIELDS subsection while a dynamic link may or may not use a LINKFIELDS subsection.

Static link

A static link automatically finds a matching record in the link file. You give complete matching information in the LINKFIELDS subsection of the file section by equating fields in the second file with fields in the main file. For example, in the case of `uphh.man`:

```
INPUTFILE
  InFile : InitHH ('InitHH.asc', ASCII)
 LINKFIELDS
    Region = UpFile.Ident.Region
    Stratum = UpFile.Ident.Stratum
    SampleNum = UpFile.Ident.SampleNum
```

You cannot influence the linking in the MANIPULATE section if it is static linking. The second file, the link file, must be sorted by the same fields. If the file is ASCII, this is done automatically when the file is brought into memory. By default, INMEMORY = YES is invoked if the link file is an ASCII file.

Dynamic link

Declare a dynamic link when you wish to personally take control of the matching in the MANIPULATE section. The LINKFIELDS section for a dynamic link for the preceding example is as follows:

```
LINKFIELDS
  Region
  Stratum
  SampleNum
```

If the link file is an ASCII file, the LINKFIELDS subsection is required. If the link file is a Blaise file, the LINKFIELDS subsection is not required.

To find a matching record in the link file, in the MANIPULATE section use a SEARCH and READ together, or use a GET.

Blaise files as link files

Blaise files can be link files, but the link fields have to be a primary key. If the link file is a Blaise file and there is no LINKFIELDS subsection to a file section, you can still dynamically link files through manipulations in the MANIPULATE section.

8.2.2 Day file

A day file, or log file, can be created to record the time and date that various Manipula setups were executed. This can be useful to ensure that a series of jobs were run in the correct order, or that they were run at all. Manipula adds to this file automatically as Manipula is executed.

You can write a line of text that will appear in the day file with the DAYFILE instruction. The day file is written if the setting DAYFILE is present in the global SETTINGS section or if you ask for it on the command line with the /D option. If the file is already there, information is appended; otherwise it is created. See Chapter 7 and the *Reference Manual* for more details.

8.2.3 Message file

You can use a message file to help debug Manipula setups or record other problems and events. Manipula writes system problems to the message file. For example, if there is a run-time error or a subscript error, then Manipula will write a message to this file.

You can write additional information to the message file with the MESSAGE instruction. Manipula provides a default message file name, but you can override this with the MESSAGEFILE setting or from the command line with the /R option.

See the *Reference Manual* and Chapter 7 for more information about the message file options.

When arrays are used, it is a good idea to have the global setting as follows:

```
SETTINGS
  SUBSCRIPTERROR = MESSAGE
```

This way, if you have a subscript error in written code, a note will be written in the message file.

8.2.4 Customised information files

You can create your own customised information files by using normal Manipula techniques. For example, suppose you are importing data into Blaise and you have some ability to detect bad data when it comes into the data set. You can write a message to an output file which is an ASCII text file.

8.2.5 File methods WRITE, KEEP, WRITEALL, and KEEPALL

Manipula allows you to time the writing of a file. The WRITE method (which is usually used) writes to the data file at the moment it is reached in the MANIPULATE section. Conversely, KEEP writes to the file only at the end of the MANIPULATE section. Therefore, the difference with WRITE is that the whole MANIPULATE section is executed first, so values of variables may be changed after the KEEP instruction, before the record is actually written to the file.

WRITEALL writes all files when it is reached. It has the same effect as specifying a WRITE instruction for all files separately. KEEPALL writes all files at the end of the MANIPULATE section. It has the same effect as specifying a KEEP instruction for all files separately.

8.3 Example File Structures

Data files can have different kinds of structure or storage formats. The Manipula techniques you use are dictated by the structures of the files.

If you receive data from other organisations or from other software packages, the files are often not in the format needed for subsequent processing. The following examples show a few common structures for ASCII files. They are based on name and address manipulation, a common use of Manipula. (Blaise data files can also have different structures based on blocks. See Chapter 4 for details.)

The Blaise data model `ncs07.bla` is an example of a rostered household instrument that has address information at the highest level and individual information in the person roster at a lower level. There are many ways information can appear in an ASCII file or files, as shown on the following pages.

8.3.1 Address and roster information in one file

The household address information and the person information that appear in the roster all may be in one file with the following format:

```
[ID info] [Address info] [Person 1]  ... [Person 5]
```

An example of this is in the file `initfile.asc`. Its data definition is as follows:

```
DATAMODEL InitFile {Goes with InitFile.ASC file}
  FIELDS
    Region   : INTEGER[2]     {ID information}
    Stratum  : INTEGER[4]
    SampleNum: INTEGER[4]
    Street   : INTEGER[27]    {Address information}
    Apartment: STRING[3]
    Town     : STRING[20]
    State    : STRING[20]
    PostCode : STRING[10]
    PhoneNum : STRING[10]
BLOCK BMember                 {Roster information}
  FIELDS                      {repeated 5 times}
    Age      : INTEGER[2]
    FirstName: STRING[12]
    SurName  : STRING[18]
ENDBLOCK
    Member   : ARRAY [1..5] OF BMember
ENDMODEL
```

In this file, there is physically one line in the ASCII file for each form in the Blaise file. (A programmer would say that there is one physical record per logical record in this ASCII file.)

The following examples illustrate the ways in which to deal with data in different file formats depending on where they are coming from or to where they are going. Manipula has facilities to handle all of these situations and more:

8.3.2 Address and roster information in separate files

The same information may be held in two files, one for the address information, the other for the roster information. Both files hold the necessary identification fields to allow them to be linked together. The first file of identification and address information requires only one line per Blaise form. Its format is as follows:

```
[ID info] [Address info]
```

A file that holds address information in this format is `inithh.asc`. Its definition is:

```
DATAMODEL InitHH {Goes with InitHH.ASC file}
  FIELDS
    Region   :  INTEGER[2]   {ID information}
    Stratum  :  INTEGER[4]
    SampleNum:  INTEGER[4]
    Street   :  STRING[27]   {Address information}
    Apartment:  STRING[3]
    Town     :  STRING[20]
    State    :  STRING[20]
    PostCode :  STRING[10]
    PhoneNum :  STRING[10]
ENDMODEL
```

The corresponding person roster information is usually held in one of two different file formats. The first format holds all person roster information for a Blaise form on one line of the ASCII file as illustrated in the following example:

```
[ID info] [Person 1]  . . . [Person 5]
```

An ASCII data file that holds roster information in this way is `initros1.asc`. Its definition is as follows:

```
DATAMODEL InitRost {Goes with InitRos1.ASC file}
FIELDS
    Region   : INTEGER[2]     {ID information}
    Stratum  : INTEGER[4]
    SampleNum: INTEGER[4]
BLOCK BMember
    FIELDS
      Age      : INTEGER[2]  {Roster information}
      FirstName: STRING[12]  {repeated 5 times}
      SurName  : STRING[18]
  ENDBLOCK
    Member   : ARRAY [1..5] OF BMember
ENDMODEL
```

A second format for the roster information is to have only one person's information per line of the file as illustrated in the following example:

```
[ID info] [Person 1]
 .
 .
 .
[ID info] [Person 5]  {Up to this number}
```

Since not every household will have up to five members, the actual file may have a variable number of lines per Blaise form as illustrated in the following example:

```
[ID 1] [Person 1]
[ID 1] [Person 2]
[ID 1] [Person 3]
[ID 2] [Person 1]
[ID 2] [Person 2]
[ID 2] [Person 3]
[ID 2] [Person 4]
[ID 3] [Person 1]
etc.
```

Its definition is as follows:

```
DATAMODEL InitRos1 {Goes with InitRos2.ASC file}
  FIELDS
    Region   : INTEGER[2]  {ID information}
    Stratum  : INTEGER[4]
    SampleNum: INTEGER[4]
    Age      : INTEGER[2]  {Roster information}
    FirstName: STRING[12]  {stated only one time}
    SurName  : STRING[18]
ENDMODEL
```

A file corresponding to the above definition is `initros2.asc`. (A programmer would say that there are possibly several physical records per logical record in this file.)

Alternate file structures in one file

It is possible to have address and roster information in one file with alternate formats for different types of lines. For example:

```
[ID info] [RecType] [Address info]  {record type 1}
[ID info] [RecType] [Person 1]      {record type 2}
 .
 .

[ID info] [RecType] [Person 5]  {Up to this number}
etc.
```

In this situation, you must have one data description for the address type of line and an alternative data description for the person roster type of line. In this kind of file, there must be some way to determine which data description applies to which line. This is represented by *RecType* above.

8.4 More About MANIPULATE

In this section we examine some features you can use in the MANIPULATE section.

8.4.1 Checking rules

CHECKRULES invokes all of the rules of a data model at one time for all selected forms in a Blaise data set. For example, you might want to import data from another source or you might want to apply additional edits that were not applied during data collection. This is done with a CHECKRULES setting (global or local), or a CHECKRULES method in the MANIPULATE section. The former may be used if you want to check all forms in a file, the latter if you wish to check only some forms in a file. See the setup `checkall.man` for an example.

If you want to check only part of a file and you have a secondary key that says which part of the file should be checked, then use the STARTKEY key word to skip past the part of the file that does not need to be checked.

8.4.2 Form correctness status

The FORMSTATUS method in Manipula returns the correctness status of a form. There are four correctness statuses in Blaise: *clean, suspect, dirty,* and *notchecked.* The correctness status of a form depends on the type of errors in it. Blaise has three kinds of errors: *hard*, *soft*, and *route*. If there are hard or route errors in the form, then the correctness status is *dirty*. If there are only soft errors, the form is *suspect.* If there are no errors, the form is *clean.* If the form has never been checked, then the correctness status is *notchecked.*

ERRORCOUNT

The ERRORCOUNT function returns the number of errors of a specified type in a particular form. This function is meaningful only if the FORMSTATUS of the form is other than *notchecked*. For example, if you have a file called `infile`:

```
MANIPULATE
  IF Infile.FORMSTATUS <> NOTCHECKED THEN
    NumRouteError:= ERRORCOUNT(ROUTE)
    NumHardError:= ERRORCOUNT(HARD)
    NumSoftError:= ERRORCOUNT(SOFT)
  ENDIF
```

SUPPRESSCOUNT

Blaise allows you to suppress soft errors. When a soft error is suppressed, it is no longer considered an error. Thus, the ERRORCOUNT(SOFT) will not count suppressed errors. To find the number of suppressed soft errors, use the SUPPRESSCOUNT function.

SELECTSTATUS

To select a record to process based on the correctness status, use the SELECTSTATUS setting. For example:

```
INPUTFILE
  InFile NCS07('NCS07', BLAISE)
  SETTINGS
    SELECTSTATUS = (SUSPECT, DIRTY)
```

Only the forms with the correctness status *suspect* or *dirty* will be processed.

8.4.3 Block history

Blaise tracks the history of every block, so it knows whether data in it have been changed or not. A new block always has the history *new*. You use RESETHISTORY in the MANIPULATE section to change the block's history to *unchanged*. If data in the block are changed anywhere in the Blaise system, then the history is automatically set to *changed* until RESETHISTORY is invoked again in Manipula.

The HISTORY or RESETHISTORY function can be applied to any block including the main block. For example:

```
USES
  NCS07

UPDATEFILE
  UpFile : NCS07('NCS07', BLAISE)

OUTPUTFILE
  OutFile = UpFile('NCS07OUT', BLAISE)

MANIPULATE
  IF (UpFile.HISTORY = CHANGED) THEN
    OutFile.WRITE
    UpFile.RESETHISTORY
    UpFile.WRITE
  ENDIF
```

You have to WRITE the update file in order to record the new status of *unchanged*. To do the same for a block called *Address*, use the dot notation, as shown in the following example:

```
MANIPULATE
  IF (UpFile.Address.HISTORY = CHANGED) THEN
    OutFile.WRITE
    UpFile.Address.RESETHISTORY
    UPFile.WRITE
  ENDIF
```

8.4.4 Counting forms

You can count forms in a file with the FORMCOUNT function. For example:

```
NumForms:= FORMCOUNT
```

You can also use FORMCOUNT in a condition, as shown in the following example:

```
SETTINGS
  AUTOREAD = NO

MANIPULATE
  FOR I:= 1 TO InFile.FORMCOUNT DO
    InFile.READNEXT
    . . .
  ENDDO
```

In this example, all records are read.

8.4.5 AUTOREAD = NO

Most setups require Manipula to read one record from the main file, do something, then read the next record. Since this is the most common situation, a default setting AUTOREAD = YES causes the main file to be read in this way. Because this is the default setting, you do not have to write it.

Sometimes, however, you do not want to read through the main file sequentially, or perhaps you want to cycle through it several times. In this case, you can control the reading of the main file with the global setting AUTOREAD = NO. Use a READNEXT instruction enclosed within a loop to read the main file. The setup `maketest.man` is an example of AUTOREAD = NO:

```
SETTINGS
   AUTOREAD = NO
{much code skipped}
PROLOGUE
  GlobalCounter:= 1000
MANIPULATE
  REPEAT
    InFile.READNEXT
    GlobalCounter:= GlobalCounter + 1
    OutFile.SampleNum:= GlobalCounter
    OutFile.WRITE
    IF InFile.LASTRECORD THEN
      InFile.RESET
    ENDIF
  UNTIL GlobalCounter = 2000
```

This setup will loop through all the records of the input file until 1,000 records are written in the output file. Every time the last record is reached, the input file is reset, which puts the file pointer at the top of the file. The REPEAT-UNTIL with the READNEXT controls the reading of the main file.

8.4.6 Procedures

For some demanding uses of Manipula, especially those written by Cameleon setups in which the same repetitive code is written time after time, procedures can save thousands of lines of Manipula code. Where complicated code is used twice or more, procedures can save a lot of maintenance.

There are two types of procedures in Manipula that are used for repetitive tasks: Manipula procedures and Dynamic Link Library procedures (known as DLL procedures or DLLs).

Manipula procedures

Suppose you have three assignments that are made in different places in the MANIPULATE section. They are put in a PROCEDURE before the MANIPULATE section:

```
PROCEDURE WritePerson
  UpFile.Household.Person[H].Age:=
              RosterNameFile.Age
  UpFile.Household.Person[H].FirstName:=
              RosterNameFile.FirstName
  UpFile.Household.Person[H].SurName:=
              RosterNameFile.SurName
ENDPROCEDURE
```

To use the procedure in the MANIPULATE paragraph, just use its name:

```
MANIPULATE
. . .
  WritePerson
```

To make the procedure reusable, you can use parameters. The same procedure is shown in the following example, only this time using parameters:

```
PROCEDURE WritePerson
PARAMETERS
  IMPORT HNum : INTEGER
INSTRUCTIONS
  UpFile.Household.Person[HNum].Age:=
              RosterNameFile.Age
  UpFile.Household.Person[HNum].FirstName:=
              RosterNameFile.FirstName
  UpFile.Household.Person[HNum].SurName:=
              RosterNameFile.SurName
ENDPROCEDURE
```

If this procedure is held in the file `writeper.prc`, then you can include it in the setup:

```
INCLUDE 'WritePer.Prc'
```

In the MANIPULATE section, to call a procedure with a parameter, use a parameter list:

```
WritePerson (H)
```

Dynamic Link Libraries

The use of DLLs is covered thoroughly in Chapter 5. It is also discussed in the ASCII file `manidll.rtf` in the Blaise system folder. Whether you use Manipula procedures or DLLs depends on the application at hand. Wherever possible, you should use Manipula procedures and save DLLs for things that Manipula truly cannot do. Possible uses of DLLs include:

- Obtaining information from Windows®: current directory information, current drive, or check the amount of free disk space.
- Implementing trigonometric functions.
- Reading from or writing to different databases.

8.4.7 Block computations

Manipula allows block computations between blocks that have the same data definition. Blocks have the same data definition if they have the same number of fields in the same order, each with the same type definition. The different blocks do not need to have the same field names. When a block computation is to be performed, Manipula checks to make sure that the blocks involved in the block computation are the same, including any subblocks.

You can invoke block computations between two blocks in one data model, between two blocks in different data models, or between a block in a data model and one in an AUXFIELDS section of the Manipula setup. In the setup `many2one.man`, there is a block in the input file called *IdAddress* and another identically defined block in the AUXFIELDS section called *AuxIdAddress.* To transfer data from one to another, you can use the simple assignment statement:

```
AuxIdAddress:= InFile.IdAddress
```

The one assignment above takes the place of *N* assignments, where *N* is the number of fields in the blocks. Block computations not only save a lot of coding, they speed up processing because Manipula will transfer all data in a block as one entity, not as separate fields.

8.4.8 Functions

Functions can save a lot of programming, and Manipula provides a wide range of functions. Refer to the *Reference Manual* for a list and explanations.

8.4.9 Exits from loops

When you have array processing of blocks, often some or most of the blocks do not hold data. For example, in the data model `ncs07.bla`, there is space for 20 people in the household roster. Usually, there will be five or fewer members of the household. You should not have to loop through the array 20 times if you can detect that some array blocks are not filled in.

In Manipula, for every looping construct, there is a way to leave the loop before running through all instances of a block. This is demonstrated in the Manipula setup `initboth.man`.

```
FOR I:=1 TO 6 DO
  IF InFile2.Member[I].Age <> EMPTY THEN
     (do something)
  ELSE
     EXITFOR
  ENDIF
ENDO
```

As soon as *Infile2.Member[I].Age* is empty, the loop will be exited. Over many forms, this can save a lot of looping. The efficiency of this method depends on having a field, such as *Age* above, that is always filled. If the field in the condition is empty when there are other data in the block or in succeeding blocks, then some records will be missed.

Other constructs with exiting capability are WHILE-DO, which you can exit with EXITWHILE, and REPEAT-UNTIL, which can be exited with EXITREPEAT.

8.4.10 Stopping Manipula

You can stop a Manipula setup with three instructions: READY, HALT, and PAUSE.

READY

The READY instruction stops the current MANIPULATE section. If there are further sections in the setup such as SORT, then these other sections will be executed. If there are no further sections in the Manipula setup, then the setup is stopped.

For example, if you want the first 100 records of a data set for a test, you can do the following:

```
MANIPULATE
  OutFile.WRITE
  IF InFile.RECORDNUMBER = 100 THEN
    READY
  ENDIF
SORT
  ...
```

The SORT section will be carried out on the 100 records.

HALT

The HALT instruction stops the Manipula setup regardless of whether there are further instructions or sections in the setup. You may wish to check that a value is in place for a parameter before continuing. For example:

```
PROLOGUE
  IF PARAMETER = '' THEN
    DISPLAY('PARAMTER not specified')
    HALT
  ENDIF
MANIPULATE
  OutFile.WRITE
  IF InFile.RECORDNUMBER = 100 THEN
    READY
  ENDIF
SORT
  etc.
```

In this setup, if the parameter is not set from the command line or in the Manipula run parameters, then the whole setup is halted regardless of which sections follow the PROLOGUE section. The HALT instruction can be used in the MANIPULATE section as well.

PAUSE

The PAUSE instruction stops Manipula temporarily until you click the *OK* button. If you press the *Cancel* button, you will be prompted to stop the execution of the setup.

8.4.11 Debugging Manipula setups

Some debugging methods have already been covered. For example, the message file, the day file, and any customised information file you create can all help you trace problems. The PAUSE instruction will let you inspect the Manipula progress on the screen. Another useful instruction for debugging is the DISPLAY instruction.

DISPLAY

The DISPLAY instruction allows you to display any information on the screen when the setup is executed. If you give the system the WAIT sub-instruction, it will pause until you press any key, similar to the PAUSE instruction.

```
DISPLAY('In Prologue, Address = '+ Address, WAIT)
```

In production, use braces { } to comment out the WAIT sub-instruction:

```
DISPLAY('In Prologue, Address = '+ Address {, WAIT})
```

To improve performance in production, comment out the DISPLAY instruction altogether.

Test data set

A well-conceived test data set is very useful for debugging both Data Entry Program instruments and Manipula setups.

8.5 Manipula and Its Environment

This section discusses the influence of command line parameter strings and environment variables on Manipula. It also discusses local area network issues.

8.5.1 Command line parameter strings

You can pass a parameter string to a Manipula setup from the command line, a batch file, or a Maniplus setup. The parameter string can be used to influence the behaviour of the setup. From the setup `initial1.man`:

```
MANIPULATE
  IF PARAMETER <> 'Init' THEN
```

In this setup, if the command line parameter is *Init* then only household address information is read into a Blaise data set. Otherwise, the address plus additional person-level information is read in.

To pass information to the Manipula setup from the command line, use the /P command line option. An example:

```
Manipula Initial11.MAN /PInit
```

You can also pass multiple parameters separated by a semicolon; for instance:

```
Manipula Asetup.MAN /P1996;5
```

In the MANIPULATE section, you can refer to the second parameter as follows:

```
IF PARAMETER(2)= '5' THEN
   ....
ENDIF
```

Note that the reference to a parameter is case sensitive.

8.5.2 Environment variables

Another way to pass information to a Manipula setup is to use the environment variable function ENVVAR. An environment variable holds information that can be accessed by any program.

You can use environment variables to pass information to the setup. Typically you do this in a batch file (`.bat` extension) before calling the Manipula setup. For example, a `.bat` file might look like this:

```
SET BLAISEUSER=JENNIFER
SET TASK=CADI_
CALL Manipula SETUPEXA.MAN /PInit /E
```

In this way, you can put the name *Jennifer* directly into reports or IF conditions on the type of task at hand. You must be careful with the SET function. It does not allow spaces between the variable name and the intended value.

USERNAME

The function USERNAME gives you access to the environment variable BLAISEUSER. If this environment variable has not been set, the function returns the name of the current user as determined by the operating system. Examples of USERNAME are:

```
OneLine:= 'User name: ' + USERNAME
OutFile.WRITE

IF USERNAME = 'JENNIFER' THEN
```

Environment settings in Windows® registry

You can store environment settings also in the registry. The Blaise system will also search in the registry under key HKEY_CURRENT_USER\Environment\. The system will first look for the environment variable in the environment. Only when the variable can not be located there will the registry be used.

If you want to use the environment variable BLAISEUSER then you can enter the string value BLAISEUSER in the registry under key HKEY_CURRENT_USER\Environment.

You can use the program RegEdit.exe (part of the Windows® operating system) to modify your registry.

ENVVAR

If you want to use any environment variable name, then use the function ENVVAR. This function takes a parameter, the name of the environment variable, and returns the value of that environment variable. For example:

```
ENVVAR('TASK')
```

returns the value of the environment variable, which may be something like *READIN* or *CADI_*.

Notice that you can also get hold of the environment variable BLAISEUSER through:

```
ENVVAR('BLAISEUSER')
```

8.5.3 Local area network (LAN) issues

Blaise runs well on local area networks (LANs), but there can be conflicts when you try to mix interactive processes using the Data Entry Program (DEP) and batch processes using Manipula. For example, suppose you want to run a Manipula setup on a Blaise data set but people are currently using that same data set in the DEP. The Manipula setup may have to read all forms, but some of them might be in use when Manipula gets to them. You can designate whether the data file can be shared between Manipula and the DEP. If you allow the data file to be shared, Manipula lets you either skip any forms in use or stop and wait until the form is cleared.

ACCESS

The ACCESS setting specifies whether a data file can be shared or if Manipula should have exclusive use of it. The default is for exclusive use. To allow Manipula to share a file with the DEP, use:

```
SETTINGS
  ACCESS = SHARED
```

This can be a global setting for all files, or a local setting for individual files. If you use the exclusive setting and you try to run a Manipula setup on a file already in use, Manipula will report '*Waiting for the release of a locked file*' until the file is released by the DEP.

If the ACCESS setting is for shared data files, be sure to thoroughly test the process before implementing it.

ONLOCK

If the setting ACCESS is for shared use of data files, then you must designate the behaviour of Manipula when it runs into a form already in use. Use the ONLOCK setting. As a default, ONLOCK will wait for a form to become free before continuing. To skip over a form that is being used by another application, use:

```
SETTINGS
  ONLOCK = CONTINUE
```

The appropriate value of ONLOCK depends entirely on the application and the preferences of the developer and user. If Manipula must wait for forms to be released, then some setups can take a very long time to run. If Manipula must skip forms that are in use, then your output may be incomplete.

However, ONLOCK = CONTINUE has its uses. For example, suppose you have a Computer Assisted Telephone Interviewing (CATI) data set and an edit data set, and you want to move completed CATI forms to the edit data set. Suppose you do not care if a few forms are missed initially, because you can always pick up those forms at a later time. Your concern at the moment is expediting the process.

You can consider using the STARTKEY option to skip the part of the data file that you do not want. For example, if you have a secondary key that tracks the completion status of the form, if you skip incomplete forms (including those that the interviewers are calling on right at that moment), then you can go to the part of the file you really need.

CATI Emulator (Btemula.exe)

The CATI simulation utility `btemula.exe` can help you test different scenarios with ACCESS and ONLOCK. This emulation utility can play various scripts of test interview data on a LAN. You can have several or many workstations working

with btemula and test Manipula setups manually on another workstation. With this, you can experiment with different processing scenarios to see which is best.

Concurrent tasks on one data file

It is possible to run different tasks on the same file. There is nothing wrong with having interviewers and data editors operating on the same data set in different modes of DEP use. As noted above, there are several ways to handle this, and you must experiment to see what is best for your organisation.

Avoiding concurrent DEP and Manipula use

If you want to avoid the issue of concurrent use of the DEP and Manipula altogether, then the monitor.exe tool can inform you if anyone is currently using a Blaise data set.

8.6 Reformatting Files

This section discusses how to reformat files from one record to many records, or from many records to one record. We use two ASCII to ASCII examples. The examples would work just as well for Blaise to Blaise, ASCII to Blaise, or Blaise to ASCII, but then the Blaise data models might not be defined in the USES section of the setup.

8.6.1 One physical record to many

You might want to reformat a file that has all of one form's data in one physical line to a file where several physical lines in the file are used to represent one form's data. The example Manipula program we use is one2many.man. It reformats data from this format:

```
[ID info] [Address info] [Person 1]  ... [Person 5]
etc.
```

to the following format:

```
[ID info] [Address info] [Person 1]
 .
 .
 .
[ID info] [Address info] [Person 5]
etc.
```

This type of reformatting might be appropriate for a base household survey that is followed by individual surveys. In the follow-on surveys, the sampling unit is the individual, not the household. Thus we need to copy the address information for each one so that it can be verified in each successive contact.

Steps for one record to many

The following numbered steps are used to do this:

- {1} Define the input data model and data file name.
- {2} Define the output data model and data file name.
- {3} Read a record from the input data model. This is done automatically since, by default, AUTOREAD = YES.
- {4} Write the address information to the output file. For every input record, cycle through the person blocks. For every block with data, write a new line in the output file by using a FOR-DO loop in the MANIPULATE section.

Note that each step number has a corresponding part in our sample setup and is identified in the setup by the number in brackets { }:

Example setup one2many.man

The following example illustrates a reduced version of the Manipula setup `one2many.man`:

```
USES
  {1}
  DATAMODEL InitFile {Goes with InitFile.ASC file}
    BLOCK BIdAddress
      FIELDS
        Region   : INTEGER[2]
        Stratum  : INTEGER[4]
        SampleNum: INTEGER[4]
        Street   : STRING[27]
        Apartment: STRING[3]
        Town     : STRING[20]
        State    : STRING[20]
        PostCode : STRING[10]
        PhoneNum : STRING[10]
    ENDBLOCK
    BLOCK BMember
      FIELDS
        Age      : INTEGER[2]
        FirstName: STRING[12]
        SurName  : STRING[18]
    ENDBLOCK
    FIELDS
      IdAddress : BIdAddress
      Member    : ARRAY [1..5] OF BMember
  ENDMODEL

  {2}
  DATAMODEL RsltFile {Goes with RsltFile.Asc file}
    BLOCK BIdAddress
      FIELDS
        Region   : INTEGER[2]
        Stratum  : INTEGER[4]
        SampleNum: INTEGER[4]
        Street   : STRING[27]
        Apartment: STRING[3]
        Town     : STRING[20]
        State    : STRING[20]
        PostCode : STRING[10]
        PhoneNum : STRING[10]
    ENDBLOCK
    BLOCK BMember
      FIELDS
        Age      : INTEGER[2]
        FirstName: STRING[12]
        SurName  : STRING[18]

    ENDBLOCK
    FIELDS
      IdAddress : BIdAddress
      Member    : BMember
  ENDMODEL
{1}
INPUTFILE
  InFile : InitFile ('InitFile.asc', ASCII)

{2}
OUTPUTFILE
  OutFile : RsltFile ('RsltFile.asc', ASCII)
```

```
AUXFIELDS (GLOBAL)
  GlobalSampleNum : INTEGER

AUXFIELDS
  I : INTEGER

MANIPULATE
  {3}
  { input records read in automatically due to }
  { AUTOREAD = YES                              }
  OutFile.IdAddress:= InFile.IdAddress {Block comp}
  {4}
  FOR I:= 1 TO 5 DO
    IF InFile.Member[I].Age > 0 THEN
      GlobalSampleNum:= GlobalSampleNum + 1
      OutFile.Member:=
                InFile.Member[I] {Block Comp}
      OutFile.IdAddress.SampleNum:= GlobalSampleNum
      OutFile.WRITE
    ENDIF
  ENDDO
```

The setup above reformats the data to one physical record per individual. It also assigns a sample number, or identification number, to each individual. In order to assign a sequential sample number, you must define a counter that will keep its value from one input record to another. You can define such a counter in a global AUXFIELDS section as shown with the auxfield *GlobalSampleNum* above. In the FOR-DO loop the value of *GlobalSampleNum* is incremented every time *Age* is positive.

Another feature of Manipula, the use of block computations in two places, is shown in the following example:

```
OutFile.IdAddress:= InFile.IdAddress

OutFile.Member:= InFile.Member[I]
```

8.6.2 Many physical records to one

You can also reformat a file from many physical records to one. This is more difficult, because there are an undetermined number of physical records that have to be combined into one record.

One strategy is to hold information from several records in memory until there are no more input records from that form. When that situation is reached, then the input records are written as one output record. You know a new form is reached when the identification numbers change.

Steps for many records to one

The following steps are used to reformat a file from many physical records to one. In our example, we assume that the input file is sorted so that all physical records of one form follow one another. As with our previous example, each step number has a corresponding part in our example setup and is identified in the setup by the number in brackets { }:

- {1} Define the input data model and data file name.
- {2} Define the output data model and data file name.
- {3} Define a holding place for data until it is time to read them out. Use global auxfields for this.
- {4} Define a file writing procedure that can be used twice.
- {5} Read the first record from the input file and store the information in the global auxfields.
- {6} Read another record from the input data model.
- {7} Determine if the new record belongs to a different form. If so, write the combined record from the global auxfield to the output file.
- {8} Store the new input record's data in the global auxfield. Do this whether it is the first record of a new form or an additional record of the same form.
- {9} When you arrive at the end of the input file, write the last record to the output file.

Example setup many2one.man

The Manipula setup that does this is `many2one.man`. The following example provides extracts of this setup. The first part of the setup identifies the input data model and data file name and the output data model and data file name.

```
USES
  {1}
  DATAMODEL RsltFile {Goes with RsltFile.Asc file.}
    BLOCK BIdAddress
      FIELDS
        Region   :  INTEGER[2]  {  1 -  2}
        Stratum  :  INTEGER[4]  {  3 -  6}
        SampleNum:  INTEGER[4]  {  7 - 10}
        Street   :  STRING[27]  { 11 - 37}
        Apartment:  STRING[3]   { 38 - 40}
        Town     :  STRING[20]  { 41 - 60}
        State    :  STRING[20]  { 61 - 80}
        PostCode :  STRING[10]  { 81 - 90}
        PhoneNum :  STRING[10]  { 91 -100}
    ENDBLOCK
    FIELDS
      IdAddress : BIdAddress
      BLOCK  BMember
        FIELDS
          Age      : INTEGER[2] {101 - 102}
          FirstName: STRING[12] {103 - 114}
          SurName  : STRING[18] {115 - 132}
      ENDBLOCK
      FIELDS
        Member : ARRAY [1..1] OF BMember
  ENDMODEL

  {2}
  DATAMODEL InitFile {Goes with InitFil2.ASC file.}
    BLOCK BIdAddress
      FIELDS
        Region   :  INTEGER[2]  {  1 -  2}
        Stratum  :  INTEGER[4]  {  3 -  6}
        SampleNum:  INTEGER[4]  {  7 - 10}
        Street   :  STRING[27]  { 11 - 37}
        Apartment:  STRING[3]   { 38 - 40}
        Town     :  STRING[20]  { 41 - 60}
        State    :  STRING[20]  { 61 - 80}
        PostCode :  STRING[10]  { 81 - 90}
        PhoneNum :  STRING[10]  { 91 -100}
    ENDBLOCK
    FIELDS
      IdAddress : BIdAddress
      BLOCK  BMember
        FIELDS
          Age      : INTEGER[2] {101 - 102, 133 - 134, 165 -
                                 166, 197 - 198, 229 - 230}
          FirstName: STRING[12] {103 - 114, 135 - 146, 167 -
                                 178, 199 - 210, 231 - 242}
          SurName  : STRING[18] {115 - 132, 147 - 164, 179 -
                                 196, 211 - 228, 243 - 260}
      ENDBLOCK
      FIELDS
        Member : ARRAY [1..5] OF BMember
ENDMODEL

{1}
INPUTFILE
  InFile : RsltFile ('RsltFile.ASC', ASCII)

{2}
OUTPUTFILE
  OutFile : InitFile ('InitFil2.asc', ASCII)
```

In this setup you need a global AUXFIELDS section to hold data in memory until they are ready to be printed out. The block *BIdAddress* and the block *BMember*

are defined identically to corresponding blocks used in the data models in the USES section. This will enable block computations. The AUXFIELD block *BMember* is arrayed five times and can hold up to five people's information.

```
{3}
AUXFIELDS (GLOBAL)
  BLOCK BIdAddress
    FIELDS
    {a bunch of fields here}
  ENDBLOCK
  BLOCK BMember
  FIELDS
    Age      : INTEGER[2]
    FirstName: STRING[12]
    SurName  : STRING[18]
  ENDBLOCK
  FIELDS
    InFileCounter : INTEGER
    GlobalCounter : INTEGER
    AuxRegion     : INTEGER[2]
    AuxStratum    : INTEGER[4]
    AuxIdAddress  : BIdAddress
    AuxMember     : ARRAY [1..5] OF BMember
```

You read the first data record and store its data in the auxfield blocks. You must store the first record's identification information in *AuxRegion* and *AuxStratum.* Since these are global auxfields, they will hold their values from one input record to the next and thus can be used for comparison with the next record's identification values.

The PROLOGUE section is executed only one time when the first record is read. Other than that, it is like a MANIPULATE section.

```
{5}
PROLOGUE
  AuxRegion:= Infile.IdAddress.Region
  AuxStratum:= Infile.IdAddress.Stratum
  AuxIdAddress:= InFile.IdAddress {Block compute}
  AuxMember[1]:= InFile.Member[1] {Block compute}
  GlobalCounter:= 1
```

After storing the first record's information, you read the next record. Having read the next form, you determine if its identification values are the same as those of the previous form, which are held in global auxfields.

```
{7}
IF ((AuxRegion <> InFile.IdAddress.Region) OR
    (AuxStratum <> InFile.IdAddress.Stratum)) THEN
```

If the next record's identification values are different, then fill in the values of the output record and then write the record.

```
{7}
WriteOut                              {Procedure}
```

Flush the person roster in the auxfields because they are global and would otherwise hold old, inappropriate values. Then reset the identification information *AuxRegion* and *AuxStratum* to the new values and the new address information.

```
{8}
  {Init global auxfield blocks}
  FOR I:= 1 TO 5 DO
    AuxMember[I]:= EMPTY
  ENDDO

  AuxIdAddress:= InFile.IdAddress  {Block compute}
  AuxMember[1]:= InFile.Member[1]  {Block compute}
  AuxRegion:= InFile.IdAddress.Region
  AuxStratum:= InFile.IdAddress.Stratum
```

Information is written to the auxfields whether the input record is the first record of a new form or a subsequent record of the same form. However, you have to take care to put the person roster block in the correct auxfield array block. This is done with a global auxfield counter.

```
ELSE
  {8}
  GlobalCounter:= GlobalCounter + 1
  AuxMember[GlobalCounter]:=
     InFile.Member[1] {Block Compute}
ENDIF
```

Finally, if the end of the input file is reached, write out the last output record.

```
{9}
IF InFile.LASTRECORD THEN
  WriteOut                           {Procedure}
ENDIF
```

8.7 Importing Blocks of Data Into Blaise

In Chapter 7, converting data from Blaise to ASCII and ASCII to Blaise was covered for simple cases where all data of the form are read in or out, the data model is not very big, and there is little or no hierarchy in the data model. In these cases, the Manipula setup is very simple, consisting of only a few lines. Chapter 7 further discussed how to condition the data conversion based on values found in the data set. For example, it was shown how to read out only complete forms, or how to read complete forms to one file and incomplete forms to another.

When you have large or hierarchical data models, you need more control of where and how data are to be placed in the output data set. This is true whether the output data set is Blaise or ASCII. For example, some data models consist of thousands of potential questions, nested in hierarchies of blocks. If you were to read data out using the techniques in Chapter 7, which Blaise and Manipula will allow you to do, you would have one extremely long data record per form. Usually, the data analysis software you use will not be able to handle such a long data record.

You probably want some structure in the output data record. For example, you might want all data collected in tables to be output in the same manner as they appear on the screen, one row on top of another. To do this, use the block structure of the data model.

If the survey is a follow-up to a previous one, or if you have name and address information from your sampling frame, you might want to import the information into the data files before starting the survey. For a hierarchical data model (such as `ncs07.bla`), you import address information at the highest level and person information into a lower level roster (table).

8.7.1 Address and roster information in one file

The simplest situation is where the household address and person data are all held in one file and all information for one Blaise form is held on one line in the ASCII file:

```
[ID info] [Address info] [Person 1]  ... [Person 5]
```

An example of this is in `initfile.asc`. The Manipula setup that reads in the data is `initial.man`. In this example, the input file does not have sample

numbers assigned. In order to assign a sequential sample number during read-in, declare a global auxfield:

```
AUXFIELDS (GLOBAL)
  GlobalCounter : INTEGER
```

In the MANIPULATE section, increment the counter *GlobalCounter* by *1* for every input record.

Associate fields in the Blaise data model with fields in the ASCII file in the MANIPULATE section. The following is for household address information:

```
MANIPULATE
IF Region <> EMPTY THEN
  GlobalCounter:= GlobalCounter + 1
  OutFile.Ident.Region:= Region
  OutFile.Ident.Stratum:= Stratum
  OutFile.Ident.SampleNum:= GlobalCounter
  OutFile.Address.Street:= Street
etc.
```

Since blocks between the data models are not identically defined, block computations are not possible.

Because the person data are held in an array of blocks in both the Blaise data model and the ASCII data file, handle these assignments in a FOR-DO loop.

```
FOR I:= 1 TO 5 DO
  IF InFile.Member[I].Age <> EMPTY THEN
    Outfile.Household.Person[I].Age:=
                      InFile.Member[I].Age
    Outfile.Household.Person[I].FirstName:=
                      InFile.Member[I].FirstName
    Outfile.Household.Person[I].SurName:=
                      InFile.Member[I].SurName
    OutFile.Address.HHSize:= I
  ELSE
    EXITFOR
  ENDIF
ENDDO
```

The computations in the array will be done only if the *Age* field in the *Ith* roster element in the ASCII file is not empty.

8.7.2 Address and roster information in separate files

If the address and roster information is in separate files, you can use different strategies to read both files into the Blaise data file. You can:

- Write a Manipula program to combine the two ASCII files into one ASCII file of the format used above in `initial.man`. Then use `initial.man` to read data in.
- Import the household data first, then import the roster information. When you import the roster file, update the existing Blaise file with an UPDATEFILE section in the second Manipula setup. Link the files together in the Manipula setup using the common identification values in the Blaise and ASCII files.
- Import the household data and the roster data at the same time from the two ASCII files. Link files together in the Manipula setup using the common identification values in the two ASCII files.

The first method is the more difficult one, and since you can handle this situation with other techniques, it is not recommended. The second method is necessary if you already have a Blaise file and you want to add data to it. The second method is a little more difficult than the third method.

The third method of linking files in one Manipula setup is probably easiest, because only one Manipula setup is necessary.

In the following sections, we demonstrate the second and third methods.

8.7.3 Two-stage ASCII read-in with UPDATEFILE

In this example, there is an ASCII file of address data that are to be read in at the highest level of the data model, and a second ASCII file of data to be read in at a lower level into a roster (table) in the Blaise instrument.

First, import the household address information. This is done with Manipula setup `inithh.man.`

In this setup you have assignments in the MANIPULATE section of the form:

```
OutFile.Ident.Region:= InFile.Region
OutFile.Address.Apartment:= InFile.Apartment
```

Once invoked, the Manipula setup `inithh.man` has initialised the Blaise data set. At this point, the Blaise data set contains only some household-level address information. We now want to add the person-level data in the roster.

Two roster structure situations

There are two situations to consider. Both situations have the same household file:

```
[ID info] [Address info]
```

The situations differ in the structure of the second ASCII person roster file.

In the first situation, the file has the form:

```
[ID info] [Person 1]  . . . [Person 5]
```

In the second situation, the file has the form:

```
[ID info] [Person 1]
 .
 .
 .
[ID info] [Person 5]  {Up to this number}
```

In this case, you do not know ahead of time how many records in the person ASCII file belong to the same Blaise form. This number will vary as you go from one household to another.

In either case, since the person data are to be added to the already existing Blaise data set, an UPDATEFILE section is needed.

```
UPDATEFILE
  UpFile : NCS07 ('NCS07', BLAISE)
```

Situation 1: One line in the ASCII file corresponds to one Blaise form

In the first situation, all person information for one Blaise form is on one line of the ASCII file. The setup for this is `initros1.man`, which reads in `initros1.asc`. The INPUTFILE section for this setup is as follows:

```
INPUTFILE
  RosterNameFile : InitRost ('InitRos1.asc', ASCII)
LINKFIELDS
  Region = UpFile.Ident.Region
  Stratum = UpFile.Ident.Stratum
  SampleNum = UpFile.Ident.SampleNum
```

Since there is a one-to-one correspondence between a record in the Blaise data set and the ASCII file, we can make an easy link between the two files. The LINKFIELDS subsection of the INPUTFILE section associates link fields between the two files. When Manipula reads a record from the first listed file, which is the master file, it will automatically create a link with the second file. There is no need to use file searches and reads in the MANIPULATE section. This automatic linking is known as *static linking.* Since the link between the files is done automatically, all that remains is to cycle through the person blocks and write to the update file, the Blaise file:

```
MANIPULATE
  FOR I:= 1 TO 5 DO
    IF RosterNameFile.Member[I].Age <> EMPTY THEN
      HHSize:= HHSize + 1
      Household.Person[I].Age:=
              RosterNameFile.Member[I].Age
      Household.Person[I].FirstName:=
              RosterNameFile.Member[I].FirstName
      Household.Person[I].SurName:=
              RosterNameFile.Member[I].SurName
    ENDIF
  ENDDO
  Address.HHSize:= HHSize
  UpFile.WRITE
```

The household size field, *HHSize*, is calculated based on data in the ASCII file.

Situation 2: Several ASCII records per one Blaise form

In the second situation, the input ASCII file has up to several lines per Blaise form. The setup we use is `initros2.man`. One way to handle this is to read in one form at a time from the update file and then search the input file for a matching record. Once the matching record is found, the second file should be accessed repeatedly until there are no more matching records. When records no

longer match, write out the update file record. The INPUTFILE section will look like this:

```
INPUTFILE
  RosterNameFile : InitRos2 ('InitRos2.asc', ASCII)

LINKFIELDS
  Region
  Stratum
  SampleNum
```

In order to perform a SEARCH in the MANIPULATE section, you have to declare link fields in the INPUTFILE section. Locating records using the SEARCH or GET functions is known as *dynamic linking.* At this point, you do not need to associate link fields with update file fields because this will be done in the MANIPULATE section. The MANIPULATE section looks like this:

```
MANIPULATE
  Reg:= UpFile.Ident.Region
  Stra:= UpFile.Ident.Stratum
  SampN:= UpFile.Ident.SampleNum
  IF RosterNameFile.SEARCH(Reg, Stra, SampN) THEN
    RosterNameFile.READ
    H:= 1
    WritePerson                       {Procedure}
    FOR H:= 2 TO 6 DO
      RosterNameFile.READNEXT
      IF RosterNameFile.RESULTOK THEN
        IF NOT (RosterNameFile.SampleNum =
          UpFile.Ident.SampleNum) THEN
          EXITFOR
        ENDIF
      ENDIF
      WritePerson                     {Procedure}
    ENDDO
  ENDIF
  UpFile.WRITE
```

For each record in the update file (which is the master file, since it is listed first), a SEARCH is performed in the ASCII file. If the search is successful, data are copied into the update record in memory. Once the cycling is done, the update record is written to a file on disk. In the MANIPULATE section, the procedure *WritePerson* is invoked twice. It is defined in a PROCEDURE section just above the MANIPULATE section.

```
PROCEDURE WritePerson
  UpFile.Household.Person[H].Age:=
                   RosterNameFile.Age
  UpFile.Household.Person[H].FirstName:=
                   RosterNameFile.FirstName
  UpFile.Household.Person[H].SurName:=
                   RosterNameFile.SurName
ENDPROCEDURE
```

You do not need to use the above PROCEDURE, but it is good programming practice because it reduces repetitive code and is easier to maintain.

8.7.4 Reading in two ASCII files at the same time

In the examples above, the data from the two files were imported in stages. The household data were read in first, followed by the person data. However, you can read data in from both files at the same time by using link fields. Our example here only applies when there is one record in the second file that corresponds to a record in the first file.

In this type of setup (`initboth.man`), you need two input files:

```
INPUTFILE
  InFile1 : InitHH ('InitHH.asc', ASCII)

INPUTFILE
  InFile2 : InitRos1 ('InitRos1.asc', ASCII)

LINKFIELDS {Subection, not a LOCAL setting}
  Region    = InFile1.Region
  Stratum   = InFile1.Stratum
  SampleNum = InFile1.SampleNum
```

The first input file is the master file. In this setup, once a record is read from the first input file, a link is established with a record in the second input file. Since the link is with an ASCII file, the setting INMEMORY=YES is not needed.

This time, instead of an UPDATEFILE section, use an OUTPUTFILE section.

```
OUTPUTFILE
  OutFile : NCS07 ('NCS07', BLAISE)
```

The MANIPULATE section is just a series of assignments, with a FOR-DO loop to cycle through all of the person blocks.

```
MANIPULATE
  {InFile1}
  OutFile.Ident.Region:= InFile1.Region
  OutFile.Ident.Stratum:= InFile1.Stratum
  OutFile.Ident.SampleNum:= InFile1.SampleNum
  OutFile.Address.Street:= InFile1.Street
  ... {Many more statements}
  OutFile.Address.PhoneNum:= InFile1.PhoneNum
  {InFile2}
  FOR I:= 1 TO 6 DO
    IF InFile2.Member[I].Age <> EMPTY THEN
      Outfile.Household.Person[I].Age:=
                        InFile2.Member[I].Age
      Outfile.Household.Person[I].FirstName:=
                        InFile2.Member[I].FirstName
      Outfile.Household.Person[I].SurName:=
                        InFile2.Member[I].SurName
    ENDIF
  ENDDO
  OutFile.Proxy:= No
  OutFile.WRITE
```

Note that *InFile1* is sometimes used and *InFile2* is used at other times. The data definitions of the input files do not match those of the output file. Thus, you cannot use block computations. This is the reason that, in this example, all computations are field-level computations. You can rearrange the ASCII data to meet the data definition of the output file. If you do so, then block computations in the setup can be used.

Alternative record types in one input file

As mentioned above, you can have all address and roster information in one file, but with different formats for the type of line. For example:

```
[ID info] [RecType] [Address info]
[ID info] [RecType] [Person 1]
 .
 .
 .
[ID info] [RecType] [Person 5]  {Up to this number}
etc.
```

A Manipula setup that can read in this type of ASCII file is `initaltr.man`, which reads in data from `initaltr.asc`. Here you have two data model descriptions for the one input ASCII file in the USES section, and only one INPUTFILE section:

```
SETTINGS
  AUTOREAD=NO

USES
  {1}
  DATAMODEL MetaHH {Goes with the first type of record for households}
    FIELDS
      Region   : INTEGER[2]
      Stratum  : INTEGER[4]
      SampleNum: INTEGER[4]
      RecType  : INTEGER[1]
      Street   : STRING[27]
      Apartment: STRING[3]
      Town     : STRING[20]
      State    : STRING[20]
      PostCode : STRING[10]
      PhoneNum : STRING[10]
  ENDMODEL
  {2}
  DATAMODEL MetaPers {Goes with the second type of record for people}
    FIELDS
      Region   : INTEGER[2]
      Stratum  : INTEGER[4]
      SampleNum: INTEGER[4]
      RecType  : INTEGER[1]
      Age      : INTEGER[2]
      FirstName: STRING[12]
      SurName  : STRING[18]
  ENDMODEL

{1 & 2}
INPUTFILE DataHH : MetaHH ('Initaltr.asc')
ALTERNATIVE
  DataPers : MetaPers
```

The INPUTFILE section has an ALTERNATIVE subsection. This indicates that sometimes the data model `datahh` is used to describe the alternate types of lines in the same input file, and at other times the data model `datapers` is used. The field *RecType*, which is the 11th column in both data files, indicates which record type definition should be used. The MANIPULATE section uses the record type in an IF condition to know what to do:

```
IF DataHH.RecType = 1 THEN
  {assignments with data model DataHH}
ELSE
  {assignments with data model DataPers}
ENDIF
```

The format of the input file is another example of many physical records belonging to one logical record. You have to hold the values of the output record in memory until all appropriate input records have been read. In this setup, data are assigned directly to the output file from the record of the input file.

You have to prevent Manipula from resetting the values of the output record every time an input record is read. Do this with the setting INITRECORD = NO, which goes with the OUTPUTFILE section. Thus you take care of initialising the output record in the MANIPULATE section.

```
{3}
OUTPUTFILE
  OutFile : NCS07 ('NCS07', BLAISE)
SETTINGS
  INITRECORD = NO
```

You initialise the output record in the MANIPULATE section with the INITRECORD file method. It is appropriate to do this after you have detected another form's data in the input file and after the output record has been written.

```
  WriteForm
  OutFile.INITRECORD
```

The same goal was accomplished in the setup `many2one.man` by using global auxfields to hold output data until it was time to write them.

8.8 Exporting Blocks of Data from Blaise

This section discusses two types of data export: exporting blocks of data from a Blaise data set in a form suitable for a relational database, and exporting one or a few blocks of data to another package. Both techniques use a file type called *ASCIIRelational*. A metadata description of the blocks of data is produced as well.

8.8.1 ASCIIRelational file types

ASCIIRelational files are ASCII files that facilitate the interchange of data between a Blaise data set and a relational database. In a relational database, data for one overall entity are held in two-dimensional tables. For example, a main table can hold names of companies. For each company in the main table, another table can hold locations of the company. In the second table of locations, there can be one or more records relating back to one entry in the main table of company names.

For this example, the company table is known as the *parent* table, and the location table is the *child* table of that parent. Each row in the child table has a pointer to the corresponding row in the parent table, and more than one child row can point to the same parent row.

The way Blaise instruments are structured with blocks bears some resemblance to the way relational databases store data. ASCIIRelational file types export data for every block with independently defined data storage into its own file. Thus if your data model has 50 blocks, each with independent data storage, then the ASCIIRelational readout will produce 50 files. Usually you do not want that many files, so Blaise gives you ways to reduce the number of files that are output and to make them correspond to what the relational database is expecting.

In our example above, each row in the company table has pointers to the appropriate rows in the location table. This is the reverse of the direction for a relational database. Therefore, before a direct mapping between the ASCIIRelational files and a relational database can be made, there have to be conversions.

When you export data from Blaise using ASCIIRelational, an ASCII file is produced for every block in Blaise with independently defined data storage. Each ASCII file corresponds to a table in the relational database. The Manipula setup to export data in ASCIIRelational form is very similar to the setup for exporting ASCII data and is easily created using the Manipula Wizard:

```
SETTINGS
  DESCRIPTION = 'BLAISE TO ASCIIRELATIONAL'

USES
  InputMeta 'ModelName'

INPUTFILE InputFile1: InputMeta ('BlaiseBD', BLAISE)

OUTPUTFILE OutputFile1: InputMeta ('AsciiRel', ASCIIRELATIONAL)

MANIPULATE
  OutputFile1.WRITE
```

Note that you do not specify a file extension for the ASCIIRelational file.

8.8.2 EMBEDDED and ordinary blocks

There are two ways in which blocks can be declared that affect the data exported from the Blaise database:

- Independent blocks (not embedded)
- Dependent blocks (embedded)

Consider the situations in the following data model. The situations are noted in { } brackets in the code.

```
{Situation 1, independently definded block type}

BLOCK BPerson
ENDBLOCK
FIELDS
  Person : BPerson

{Situation 2, embedded, dependently defined block type}

EMBEDDED BLOCK BPerson
ENDBLOCK
FIELDS
  Person : BPerson

{Situation 3a, subblock Person is independently defined from Situation 1
above}

TABLE THouseHold
  FIELDS
   Person : ARRAY [1..20] OF BPerson  {from situation 1}
ENDTABLE
FIELDS
  HouseHold : THouseHold

{Situation 3b, subblock Person is dependently defined (embedded) from
Situation 2 above}

TABLE THouseHold
  FIELDS
   Person : ARRAY [1..20] OF BPerson  {from situation 2}
ENDTABLE
FIELDS
  HouseHold : THouseHold
```

```
{Situation 4}

TABLE THouseHold
  BLOCK BPerson              {independent block}
  ENDBLOCK
  FIELDS
    Person : ARRAY[1..20] OF BPerson {20 instances}
ENDTABLE
FIELDS
  HouseHold : THouseHold

{Situation 5}

TABLE THouseHold
  EMBEDDED BLOCK BPerson    {embedded (dependent)
                                     block type}
  ENDBLOCK
  FIELDS
    Person : ARRAY [1..20] OF BPerson
              {20 instances of embedded block type}
ENDTABLE
FIELDS
  HouseHold : THouseHold

{Situation 6}

TABLE THouseHold
  BLOCK BPerson              {independent block}
  ENDBLOCK
  FIELDS
    Person : BPerson                 {1 instance}
ENDTABLE
FIELDS
  HouseHold : ARRAY [1..20] OF THouseHold
```

In situations 2, 3b, and 5, the block type *BPerson* is defined as part of the surrounding block through the use of the key word EMBEDDED. In both Blaise and its ASCIIRelational form, the blocks of data are stored as part of the surrounding block. In situation 2, the surrounding block is the data model itself. In situations 3b and 5, the surrounding block is the table *THouseHold*.

Situation 3a differs from situation 4 only in where the block *BPerson* is defined. In both, data for the block type *BPerson* are stored separately from the surrounding block's data. However, you can modify situation 4 and store *BPerson*'s data with the surrounding block by using the key word EMBEDDED as in situation 5. You can do the same with situation 3a if the block *BPerson* has the EMBEDDED key word, even if it is defined in a different place in the data model.

When data are read out, each non-embedded block's data will be read out to a separate file. There is enough information to maintain referential integrity for a relational database that may hold these data. The format of the ASCIIRelational files that are produced is:

```
[Key] [InstanceNumber] [data [Sub-InstNumber] data]
```

Data from subblocks are stored separately unless they are embedded. If they are not embedded, then the parent block will record a block sub-instance number for that block in order to maintain referential integrity.

If data are read out using ASCIIRELATIONAL from the preceding situations, the following ASCII data sets are produced:

Situation 1

An ASCII file is produced for the block type *BPerson*. There will be up to one line per form. For example:

```
[Key 1] [BPerson data]
```

Situation 2

Data from the block type *BPerson* will be read out with data from the data model level. For example:

```
[Key 1] [Data] [BPerson data] [Data]
```

Situations 3a and 4

A file is produced for the block type *BPerson* separately from the file produced for the table type *THouseHold*. There can be up to 20 lines per Blaise form in the ASCII file for block type *BPerson*. The format for the household data (the numbers 1 to 20 are block sub-instance numbers) is:

```
[Key 1] [THouseHold data] [1 ... 20]
```

and for the person data:

```
[Key  1] [BPerson data]
 . . .
[Key 20] [BPerson data]
```

In the latter file, empty lines are not stored. In the first files, the block sub-instance numbers are empty in that case.

Situations 3b and 5

Data from block type *BPerson* are read out with data from the table type *THouseHold*. The format is:

```
[Key 1] [THouseHold data] [BPerson] . . . [BPerson]
```

In this situation, space is held for all 20 instances of *BPerson* even if they do not all hold data.

Situation 6

Data from block type *BPerson* are read into a separate file as in situations 3 and 4. Data from the surrounding household block will be read out as:

```
[Key  1] [THouseHold data]
 . . .
[Key 20 ][THouseHold data]
```

ASCIIRelational file name extensions

By default, each block of data that is read out will be held in its own file. The file name for each file of block data has the extension `A??`, where `??` represents a number starting with *01* (for example, `ncs07.a01`). If there are more than 99 output files, then the extension starts with `B??`. Which number a file gets in its extension depends on where it is defined in the data model.

Block metadata

The Cameleon metadata utility can produce descriptive information for each output block. See Chapter 9 for details.

8.8.3 Exporting one or a few blocks of data

The easiest way to export one or a few individual blocks of data from Blaise is to use the ASCIIRelational file type with the INCLUDEBLOCKS setting. For example, to export just the block of address information, you would use the following setup:

```
USES
  NCS07

INPUTFILE
  InFile : NCS07 ('NCS07', BLAISE)

OUTPUTFILE
  OutFile : NCS07 ('NCS07', ASCIIRELATIONAL)
SETTINGS
  INCLUDEBLOCKS = (BAddress)

MANIPULATE
  OutFile.WRITE
```

Note that the INCLUDEBLOCKS = (BAddress) refers to the type name of the block, not to the block field name *Address*. This setup name is `addrsout.man`. In the data model `ncs07`, there is only one address block for each form. The file `ncs07.a01` will include one line of address information from each Blaise form.

Sometimes a block is used more than once in a data model (there are multiple instances of a block). For example, a household roster might use the block type *BPerson* up to 20 times in each form. The Manipula setup `persnout.man` will read out data for each instance of the block type *BPerson* for which there are data in the form:

```
USES
  NCS07

INPUTFILE
  InFile : NCS07 ('NCS07', BLAISE)

OUTPUTFILE
  OutFile = InFile ('Person', ASCIIRELATIONAL)

SETTINGS
  INCLUDEBLOCKS = (BPerson)
```

Use the type identifier of the block *BPerson*, not the block field name *Person[I]*.

EXCLUDEBLOCKS

To export all but just a few blocks of data, you can use ASCIIRELATIONAL with the key word EXCLUDEBLOCKS.

Other block export techniques

It is possible to use other Manipula techniques to read out blocks of data. A schematic of how to do this with the file type ASCII instead of ASCIIRelational is shown:

```
USES
  NCS07
  IdAddress  {A Blaise data model with just the
             address block and primary key fields}
INPUTFILE
  InFile : NCS07 ('NCS07', BLAISE)

OUTPUTFILE
  OutFile : IdAddres ('IdAddres.Asc', ASCII)

MANIPULATE
  OutFile.Ident:= InFile.Ident      {block compute}
  OutFile.Address:= InFile.Address  {block compute}
  OutFile.WRITE
```

By making a separate Blaise data model with just the *Ident* and *Address* block, you can use Cameleon to produce the needed metadata. If you define the data model `address` within the Manipula setup, you can still read out the data, but then you do not have an automated way to produce the metadata.

ASCIIRELATIONAL

To import selected blocks of data into a Blaise data set without destroying other blocks of data, use UPDATEFILE. You can use ASCIIRELATIONAL just as you can use it for reading out data. The key words INCLUDEBLOCKS and EXCLUDEBLOCKS cannot be used when reading in ASCIIRelational data.

8.9 Miscellaneous Uses of Manipula

In this section, we mention a few more ways to use Manipula setups.

8.9.1 Making a test data set

When producing an application, you must test it. After basic testing is done on the instrument itself, you should give a volume test. Suppose you have made up some test interviews. You can replicate the few you have made, taking care to give each of the new copies unique identification numbers (primary keys). The Manipula

setup `maketest.man` shows how this is done. It loops through an initial data set of 18 names many times. It is an example of the use of AUTOREAD = NO.

8.9.2 Creating a library file for classify

Hierarchical coding is done with the CLASSIFY feature of Blaise. The syntax of the coding file is awkward. If you have a source file of commodity names (or, in the case of `ncs07`, cars), then you can use Manipula to convert the information into the proper syntax. Once done, the Manipula setup can correctly produce the library file very quickly, even for tens of thousands of records. The example Manipula setup `carclass.man` can be found in `\Doc\Chapter5` in the Blaise system folder.

8.10 Performance Issues

If you have large numbers of records in a data file, a very large data model, or both, the speed of Manipula can become a concern. There are several ways to speed up Manipula when it has a large task to carry out.

8.10.1 Improving performance with Manipula features

When Manipula executes a setup, it has default ways of behaving that make it very easy to write setups for common tasks. Sometimes the default behaviours can slow down processing for demanding applications.

If you do not need one of these automated features, you might be able to speed up processing. For example, you do not always want to connect data models, or automatically copy data from one to the other. These default behaviours are explained in the following table:

Figure 8-1: Default behaviours in Manipula

Default Behaviour	Performance Comment
A data tree, a representation of the data corresponding to the structure of an entire data model, is constructed in memory.	Constructing the data tree for large data models takes a lot of time during initialisation. Sometimes you might not need to construct the whole data tree. If you are going to read just part of a data model, use FILTER.
The data tree is constructed for the input and output data models separately.	If the input and output data models have the same definition, you can use the same data tree for both. See the following section on *Data sharing*.
Data are read into memory, from disk, one record at a time, from the input data file.	Input and output back and forth between disk and memory is expensive. Bring the files into memory with INMEMORY = YES.
All records, from first to last, are processed.	You don't always need to plough through all records. See STARTKEY.
If there are two or more data models, a connecting scheme connects identically named and defined fields between data models.	For large data models, the connecting algorithm can take a lot of time during initialisation. In certain circumstances, you can turn this off with CONNECT = NO.
In memory, data are copied automatically from the first data tree to the second before manipulation.	Sometimes you don't want this automatic copy before manipulation. If not, use AUTOCOPY = NO.
If there is a MANIPULATE section, then data manipulations are carried out on the output data record while in memory.	Some tricks in the MANIPULATE section can speed up the setup, especially for loops. See EXITFOR and other exiting commands.
Output data are written to the output file on disk and the process starts anew.	Here again, you can use INMEMORY = YES.

8.10.2 Skipping to a secondary key value

Secondary keys in Blaise help you to access certain parts of the data file. For example, you might have defined a secondary key called *CompletionStatus* with values:

- *Blank.* No action on this form yet
- *Incomplete.* Started but not yet complete
- *Complete.* Form is ready for further processing
- *Read_out.* Form was previously read out.

Suppose you have hundreds or thousands of forms in a data file and you want to read out only the complete forms. Normally, Manipula would churn through all forms in the data set. You can cause Manipula to skip right to the complete records and then to stop processing altogether once it reaches records that were previously read out. To use the secondary key in this way, use local settings with the input file as follows (`toascii3.man`):

```
USES
  BlaiseMeta 'NAMEJOB1'

INPUTFILE
  BlaiseData : BlaiseMeta('NameJob1', BLAISE)
SETTINGS
  KEY = SECONDARY
  STARTKEY = (Complete)

OUTPUTFILE
  AsciiData = BlaiseData('NameJob2.asc', ASCII)

MANIPULATE
  IF BlaiseData.CompletionStatus = Complete THEN
    AsciiData.WRITE
  ELSE
    READY
  ENDIF
```

The instruction KEY = SECONDARY causes the data file to be processed in the order of the secondary key *CompletionStatus*. The STARTKEY = (COMPLETE) causes Manipula to skip to the first record with that value.

In the MANIPULATE section the key word READY causes Manipula to cease processing once the condition is not satisfied.

To set up this example, read data into the data model `namejob1` with the Manipula setup `frmascii.man`. This will read in data where 10 records have *Complete* = 1 and 4 records have *Complete* = blank. When you read out with the Manipula setup `toascii3.man`, only 10 records are processed because 4 records are skipped where the field *Complete* is blank.

Declaring a number of secondary keys

There is no limit to the number of secondary keys you can declare, but they are expensive in terms of storage space. If you ever have to rewrite a data set with the Hospital utility, having a lot of secondary keys will slow the process. We suggest that you plan the use of secondary keys ahead of time. Three to five well-thought-out secondary keys will usually do for even the most demanding surveys.

8.10.3 Data sharing

When an INPUTFILE, OUTPUTFILE, UPDATEFILE, or TEMPORARYFILE use the same data model definition from the USES section, you can use the concept of *data sharing* to speed up Manipula initialisation. Normally, Blaise will build a separate data tree in memory for every file mentioned in the setup. Then, when the main record is read in, an automatic connection process will copy data from the first data tree to the other (where there are matching fields). If the data files have the same structure, you can prevent the system from building two trees, and connecting data points and copying data. This is done with *data sharing*. From the example above:

```
OUTPUTFILE
  AsciiData = BlaiseData('NameJob2.asc', ASCII)
```

The phrase *Asciidata = BlaiseData* says that the file `asciidata` shares the same structure as the file `blaisedata` and that they should share the same data tree in memory. Thus, you avoid the need for the auto-connection and auto-copy processes. This speeds up both initialisation and processing.

Cautions with computations

When you have data sharing, there is only one copy of the data in memory. Be careful with your assumptions about the values of the fields when you make computations. Once a field has been changed to a different value, then subsequent computations, such as IF conditions and the like, will be executed with that changed value, not the original value of that field. This is because the input form and the output form are in the same location in memory. However, the original value of the field will still be in the original input file on disk. Compare with UPDATEFILE above.

8.10.4 Filters

Sometimes you do not need to read a whole Blaise form into memory. For example, if you want to produce a survey management report from the data model `ncs07`, you might need information from the blocks *Manage, Ident,* and *DateTime*. If you want to list all sample numbers with a household's address, you need only the blocks *Ident* and *Address*. You do not need to bring the rest of the data model into memory. You can filter what is brought into memory with the FILTER subsection of the main input file.

The FILTER saves time in two ways. First, it builds only the data tree necessary to hold the stated blocks. Second, fewer data are manipulated. For example, from hhlist.man:

```
INPUTFILE
  InFile : NCS07 ('NCS07', BLAISE)
  FILTER
    Ident
    Address
```

When you use FILTER you cannot make reference to any other blocks in the MANIPULATE section, since they are not in memory.

```
FOR I:= 1 TO 6 DO
  IF InFile2.Member[I].Age <> EMPTY THEN
    {do something}
  ELSE
    EXITFOR
  ENDIF
ENDDO
```

As soon as *InFile2.Member[I].Age* is empty, the loop will be exited. Over many forms, this can save a lot of looping. The efficiency of this method depends on having a field, such as *Age*, that is always filled. If the field in the condition is empty when there are other data in the block or in succeeding blocks, then some records will be missed.

Other constructs with exiting capability are WHILE-DO, which you can exit with EXITWHILE, and REPEAT-UNTIL, which can be exited with EXITREPEAT.

8.10.5 TEMPORARYFILE

Using TEMPORARYFILE to hold an intermediate data set can save processing time if it is small enough to fit in memory. See the beginning of this chapter for more information on TEMPORARYFILE.

8.10.6 Block computations

Block computations save time over making many individual field-level assignments. When Manipula can copy a block of data instead of copying all of the fields of the block individually, it does not have to handle each field's value separately.

8.10.7 CONNECT = NO

When you have two or more data models in the USES section of a Manipula setup, Manipula will automatically connect fields from the first data tree in memory to the second data tree in memory for any matching fields. Sometimes you do not want this.

If automatic connecting is turned off, you can save time in two ways. First, it speeds up the initialisation process since the field connections are not made. Second, without the connect, it is not possible to automatically copy data from one data model to another, so AUTOCOPY = NO (see the next section).

If you use CONNECT = NO, you have to take care of transferring data from the input data model to the output data model in the MANIPULATE section. See the *Reference Manual* or the on-line help for details. If you know that there are no links between the input file and the output file, then set CONNECT = NO.

8.10.8 AUTOCOPY = NO

Using AUTOCOPY = NO will still allow Manipula to connect fields, but will not automatically copy data from one data tree to the next in memory. This does not save initialisation time, but it does save processing time. As with CONNECT = NO, any data transfer to the output file must be executed in the MANIPULATE section. If CONNECT = YES (the default), this can be done in the MANIPULATE section using COPY.

8.11 Example Manipula Setups

The following table lists the Manipula setup examples used in this chapter. These files can be found in `\Doc\Chapter8` of the Blaise system folder. The files are listed in alphabetical order.

Figure 8-2: Example Manipula setups

Manipula Setup	Description
checkall.man	Checks the RULES sections of a data model against forms in a data file using UPDATEFILE and CHECKRULES.
uphh.man	Shows the use of static LINKFIELDS.
one2many.man	Reformats a file that has one physical record per logical record into a file that has several physical records per logical record. Uses block computations.
many2one.man	Reformats a file that has several physical records per logical record into a file that has one physical record per logical record. Uses GLOBAL AUXFIELDS, PROCEDURES, PROLOGUE, and block computations.
initial.man	Reads in name and address information from a file that has all information for a Blaise form on one line. Uses EXITFOR.
initial1.man	Same as INITIAL except that a command line PARAMETER controls an IF condition.
initboth.man	Reads in name and address information from two files at one time. One file has address information, the other name information. Uses static LINKFIELDS.
inithh man	Reads in address information only.
initros1.man	Uses UPDATEFILE to read name information only into a Blaise data file already populated with the setup inithh. This setup handles the situation where all names for one Blaise form are on one line of the input file. Uses static LINKFIELDS.
addrsout.man	Exports just the address block from a Blaise data file using ASCIIRELATIONAL.
hhlist.man	Demonstrates the use of FILTER to read in only some data when making a report.
initaltr.man	Reads in address and name information from a file where one kind of line holds address information and another holds name information. Uses ALTERNATIVE, INITRECORD setting, INITRECORD method, and PROCEDURE.
initros2.man	Uses UPDATEFILE to read name information only into a Blaise data file already populated with the setup inithh. This setup handles the situation where only one name for a Blaise form is on one line of the input file, and where it may take several lines to hold all names for a Blaise form. Uses dynamic LINKFIELDS.
maketest.man	Makes a test data set of 1,000 forms from 18 test forms. Uses AUTOREAD = NO and the REPEAT-UNTIL with the READNEXT instruction.
persnout.man	Exports name blocks from a Blaise data file using ASCIIRELATIONAL.
toascii3.man	Exports just part of a data file depending on a secondary key. Using the STARTKEY setting, it skips past part of the data file.
writebat.man	Writes a batch file from data recorded in a Blaise instrument. Uses the TRAILINGSPACES setting.

9 Cameleon

Cameleon is a Blaise® utility that can translate descriptive information about Blaise® files into a format that can be used by other software packages. It can also be used as a diagnostic tool.

This chapter provides some programming examples, but it is only intended as an overview of Cameleon. To learn Cameleon, you need to study the example translators shipped with Blaise®, in combination with the reference material. The *Reference Manual* has a technical appendix that lists Cameleon reserved words and outlines syntax.

9.1 Cameleon and Metadata

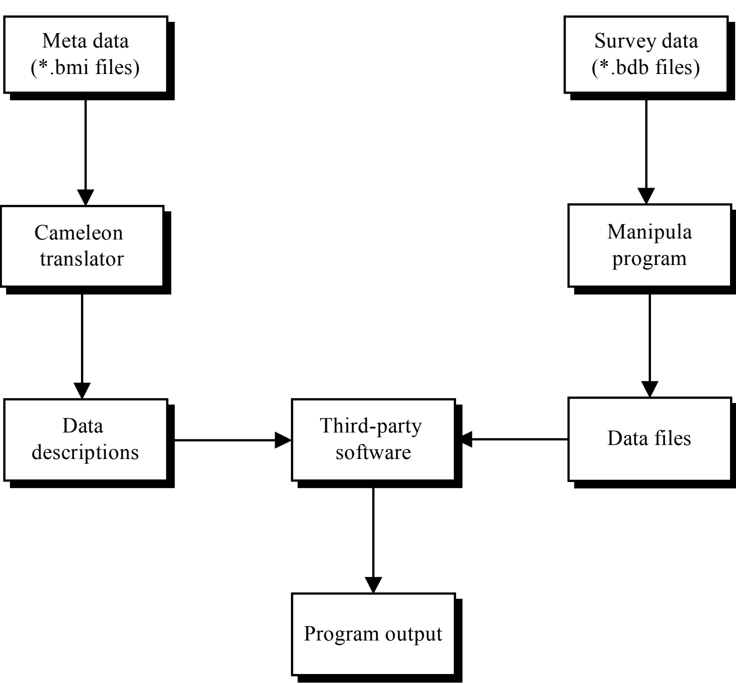
When you prepare a Blaise data model, you create a file containing information about the structure and format of the data model (the `.bmi` file), and when you enter data you create another file containing that data (the `.bdb` file). For a data model called `example.bla`, the file `example.bdb` contains the data and the file `example.bmi` contains information about the names of fields, the question text associated with each field, the codes for values accepted as data for each field, the rules, and other information. The information in the `example.bmi` file is known as the *metadata* for the `example.bla` data model. Blaise metadata files can be viewed in the Structure Browser (this is explained in Chapter 2).

You can use Cameleon to create descriptions of a Blaise data model for use by other software packages. Cameleon uses translator files to interpret the metadata descriptions in `.bmi` files to create data descriptions.

Data descriptions can also be diagnostic tools, such as the number of times a block is called in a data model. Or they can be descriptions of the format of the data used by other software, such as SPSS or SAS, to interpret ASCII data files output by Manipula.

The relationship between Cameleon and Manipula is shown in the following figure:

Figure 9-1: Cameleon and Manipula



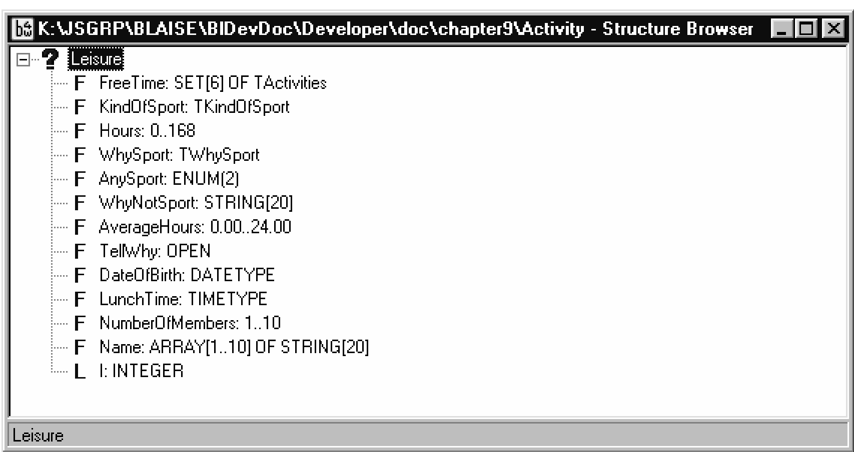
9.2 Example Data Model

The following is an example data model `activity.bla`. The RULES section has been omitted from this listing:

```
DATAMODEL Leisure "The Leisure and Sport Survey"
  TYPE
    TActivities =  (Sport "Playing sport",
                   TV "Watching TV",
                   Eating "Eating out",
                   Drinking "Going to pub or bar",
                   Sleeping,
                   Reading)
    TWhySport =    (Compete "Competition",
                   Relax "Relaxation",
                   Health "Health benefits",
                   Weight "Weight loss",
                   Other)
    TKindOfSport = (Soccer, Cycling, Tennis, Aerobics,
                   Bridge, Chess, Other)
  LOCALS
    I : INTEGER
  FIELDS
    FreeTime "What is your favorite leisure-time activity?"
        : SET OF TActivities, DK
    KindOfSport "Which is your major sporting activity?"
        : TKindOfSport
    Hours "On average, how many hours a week
        do you spend at @W^KindOfSport@W?
        @/@/@Y[INTERVIEWER] Accept an approximation."  : 0..168, DK
    WhySport "@GWhy do you engage in @W^KindOfSport?": TWhySport
    AnySport "Do you engage in any sport?" : (Yes, No), RF
    WhyNotSport "Why do you not engage in sport?": STRING[20], DK, RF
    AverageHours : 00.00..24.00, EMPTY
    TellWhy "Why do you choose to do sport?" : OPEN
    DateOfBirth "What is your birthday?
        @/@/@Y[INTERVIEWER] Be sensitive." : DATETYPE
    LunchTime "What time do you take lunch?" : TIMETYPE
    NumberOfMembers "Number of members in the household.": 1..10
    Name "Name of household member.": ARRAY [1..10] OF STRING[20]
  RULES
    …
ENDMODEL {Leisure}
```

If the data model is prepared, then the metadata file `activity.bmi` is created and can be viewed in the Structure Browser as shown in the following figure:

Figure 9-2: Structure of the data model activity.bla



Note certain characteristics of this model:

- *FreeTime* is a set of up to six activities, where the activities are declared in the TYPE section as the enumerated type *TActivities*. Each of the individual possible responses has to be differentiated.
- *KindOfSport* and *WhySport* are defined in terms of types declared in the TYPE section, and both are enumerated types.
- *TellWhy* is an open field and the length of the response will vary for each form. Usually, open responses are not input to statistical software.
- *Name* is an array of 10 elements. Each of the possible 10 individual names has to be differentiated.

9.3 Cameleon Translators Supplied with Blaise

Several translator files for various statistical packages and database systems are distributed as part of the Blaise system, together with translator files that give diagnostic information such as technical descriptions of the data model. If the supplied translator files do not meet your needs, you can modify the files or create your own translator files using the Cameleon language. The translators are listed in the following table:

Figure 9-3: Cameleon translators supplied with Blaise

Target	Description of Output	File Extension
sas	Data description for the statistical package SAS.	.sas
spss	Data description for the statistical package SPSS.	.sps
spssdot	Data description for the statistical package SPSS indicating the dot as the decimal symbol	.sps
oraclec	Setup to create Oracle database tables.	.orc
oracled	Setup to delete Oracle database tables.	.ode
paradox	Data description for the Paradox relational database system, one table for all data (using the ASCII output file parameter in Manipula).	.man .sc
paradoxf	Description for the Paradox relational database system, one table per block (using the ASCIIRELATIONAL output file parameter in Manipula).	.man .sc
toascii	Produces a Manipula setup to export data to ASCII format.	.man
asciirel	Produces a data model for each relational block.	.arm
blocstrc	Produces a tree of the block structure.	.str
questblk	Produces a tree of the field structure.	.que
dic	Produces a list of all questions in the data model.	.dic
techdesc	Produces a technical description of the data model.	.tcd
papersim	Produces a starting point for a paper questionnaire.	.pap
example	Produces a list of all blocks followed by a list of all fields with their unique names	.exa

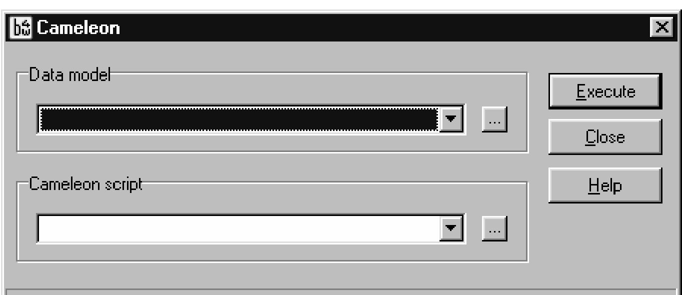
9.4 How to Start Cameleon

To start Cameleon, select *Tools* ➢ *Cameleon* from the Control Centre menu. You can also open a Cameleon script with a `.cif` extension in the Control Centre, and then select *Run* from the menu or Speedbar. You can also run the program `cameleon.exe` without opening a file first.

9.4.1 Running Cameleon

When you start Cameleon, the Cameleon window appears.

Figure 9-4: Cameleon window



- In the *Data model* box, specify the name of the `.bmi` file for the data model.
- In the *Cameleon script* box, specify the name of the Cameleon translator with a `.cif` extension.

As an example, for the data model `activity.bla`, using the SPSS Cameleon script, we would choose `activity.bmi` in the *Data model* box and `spss.cif` in the *Cameleon script* box.

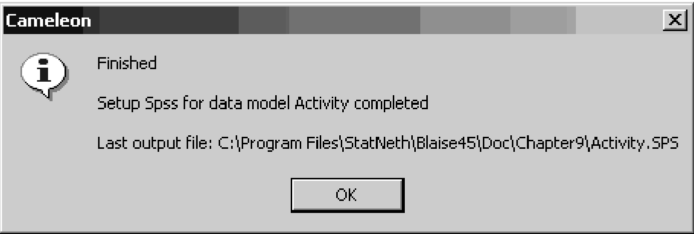
Click the *Execute* button to run the script.

If you start by opening the Cameleon file in the Control Centre, run Cameleon by selecting *Run* ➤ *Run* from the menu, or clicking the *Run* speed button. The Cameleon window will appear as shown above, and the name of the Cameleon file that you have open will already be in the *Cameleon script* box. You can then specify the `.bmi` file and click the *Execute* button.

If you have set Cameleon Run parameters and specified a data model `.bmi` file, selecting *Run* will automatically run the script without even clicking the *Execute* button in the Cameleon window. If the primary file for a project is a `.cif` file, and the project is open, selecting *Run* will run the primary (Cameleon) file for the project, *not* the file in the active window.

When Cameleon has finished running, a confirmation message appears, telling you the name of the last output file. The following sample is for the Cameleon script `spss.cif` run for `activity.bmi`:

Figure 9-5: Sample Cameleon finished message

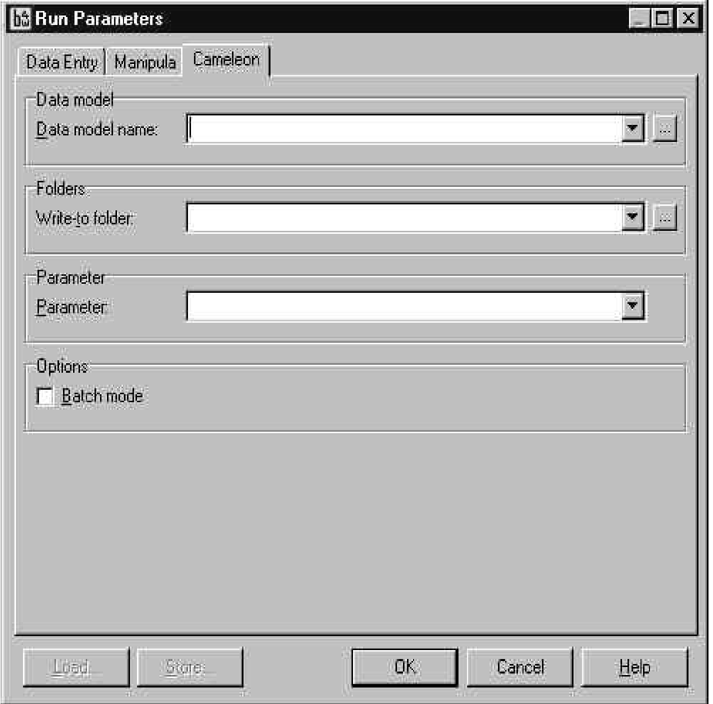


9.4.2 Setting Cameleon run parameters

You can set run parameters for Cameleon. These will affect all Cameleon files run from the Control Centre.

Select *Run* ➤ *Parameters* from the menu and the *Run* Parameters dialog box appears.

Figure 9-6: Run Parameters dialog box for Cameleon



Complete the following items as needed:

- *Data model name.* Specify the name of the `.bmi` file for the data model, or select from previous data models by clicking the down arrow.
- *Write-to folder.* Specify the name of the folder to which you want to direct your output.
- *Parameter.* Specify a value for a parameter, accessible by using the string function PARAMETER in the `.cif` file.
- *Batch mode.* Select to suppress all user questions, which ensures that no questions need to be answered.

When finished, click the *OK* button.

9.5 Cameleon Output Samples

We will look at the output from the `spss.cif` and `sas.cif` translators. In the listings that follow, the continuation symbol '_' at the end of a line means that the next line should follow directly on from the line with the '_'. For example, the two lines:

```
This is one line even though it seems _
    to be two lines
```

are equivalent to the one line:

```
This is one line even though it seems to be two lines
```

9.5.1 Output from spss.cif

The following example gives the data description of the `Leisure` data model produced for SPSS by the Cameleon translator `spss.cif`:

```
TITLE           'Leisure'.
DATA LIST FILE ='Activity.ASC' /
                FreeTim1          1 -   1
                FreeTim2          2 -   2

                FreeTim6          6 -   6
                KindOfSp          7 -   7
                Hours             8 -  10
                WhySport         11 -  11
                AnySport         12 -  12
                WhyNotSp         13 -  32        (A)
                AverageH         33 -  37
                DateOfBi         38 -  45
                LunchTim         46 -  53        (TIME)
                NumberOf         54 -  55
                Name01           56 -  75        (A)
                Name02           76 -  95        (A)

                Name10          236 - 255        (A).
MISSING VALUES
                FreeTim1        (9)
                FreeTim2        (9)

                FreeTim6        (9)
                Hours           (999)
                AnySport        (8).
COMPUTE
    #DD = TRUNC(DateOfBi / 1000000).
COMPUTE
    #RM = MOD(DateOfBi, 1000000).
COMPUTE
    #MM = TRUNC(#RM / 10000).
COMPUTE
    #YY = MOD(#RM, 10000).
COMPUTE
    DateOfBi = DATE.DMY(#DD, #MM, #YY).
FORMATS
  DateOfBi (EDATE10).
FORMATS
  LunchTim (TIME10.0).
VAR LABELS
                FreeTim1        'What is your favorite_
                     leisure-time activity?'
                FreeTim2        'What is your favorite_
                     leisure-time activity?'

                FreeTim6        'What is your favorite_
                     leisure-time activity?'
                KindOfSp        'Which is your major_
                     sporting activity?'
                Hours           'On average, how many_
                     hours a week do you spend'
                WhySport        '@GWhy do you engage_
                     in @W^KindOfSport?'
```

```
                AnySport        'Do you engage in any_
                     sport?'
                WhyNotSp        'Why do you not engage_
                     in sport?'
                AverageH        'AverageHours'
                DateOfBi        'What is your_
                     birthday?_ '
                LunchTim        'What time do you take_
                     lunch?'
                NumberOf        'Number of members in_
                     the household.'
                Name01          'Name of household_
                     member.'
                Name02          'Name of household_
                     member.'

                Name10          'Name of household_
                     member.'.
VALUE LABELS
                FreeTim1
                FreeTim2
                …
                FreeTim6        1 'Playing sport'
                                2 'Watching TV'
                                3 'Eating out'
                                4 'Going to pub or bar'
                                5 'Sleeping'
                                6 'Reading'/
                WhySport        1 'Competition'
                                2 'Relaxation'
                                3 'Health benefits'
                                4 'Weight loss'
                                5 'Other'/
                KindOfSp        1 'Soccer'
                                2 'Cycling'
                                3 'Tennis'
                                4 'Aerobics'
                                5 'Bridge'
                                6 'Chess'
                                7 'Other'/
                AnySport        1 'Yes'
                                2 'No'.
ADD VALUE LABELS
                FreeTim1        9 'Don''t Know'
               /FreeTim2        9 'Don''t Know'
               /FreeTim3        9 'Don''t Know'
               /FreeTim4        9 'Don''t Know'
               /FreeTim5        9 'Don''t Know'
               /FreeTim6        9 'Don''t Know'
               /Hours           999 'Don''t Know'
               /AnySport        8 'Refusal'.
SAVE /OUTFILE 'Activity.SAV'.
```

To use this SPSS setup:

- Export the Blaise data using Manipula as an ASCII file with the name `activity`.

- Run this setup (`activity.sps`) in SPSS with the `activity` file as the input text file.
- Use `activity.sys` as the SPSS system file.

9.5.2 Output from sas.cif

The data description of the `leisure` data model produced for SAS by the Cameleon translator `sas.cif` is given in the following example:

```
TITLE 'Leisure';

PROC FORMAT;

VALUE TE_1F
  1='Playing sport'
  2='Watching TV'
  3='Eating out'
  4='Going to pub or bar'
  5='Sleeping'
  6='Reading'
;

VALUE TE_2F
  1='Competition'
  2='Relaxation'
  3='Health benefits'
  4='Weight loss'
  5='Other'
;

VALUE TE_3F
  1='Soccer'
  2='Cycling'
  3='Tennis'
  4='Aerobics'
  5='Bridge'
  6='Chess'
  7='Other'
;

VALUE TE_4F
  1='Yes'
  2='No'
;

RUN;

DATA FILE;
INFILE 'Activity.ASC' LRECL = 455;
INPUT
    FreeTim1        1 -     1
    FreeTim2        2 -     2
    …
    FreeTim6        6 -     6
    KindOfSp        7 -     7
    Hours           8 -    10
    WhySport       11 -    11
    AnySport       12 -    12
    WhyNotSp $     13 -    32
    AverageH       33 -    37
    DateOfBi $     38 -    45
    LunchTim $     46 -    53
```

```
    NumberOf      54 -    55
    Name01 $      56 -    75
    Name02 $      76 -    95
    …
    Name10 $     436 -   455
;
LABEL
  FreeTim1   = 'What is your favorite leisure-time activ'
  FreeTim2   = 'What is your favorite leisure-time activ'
  …
  FreeTim6   = 'What is your favorite leisure-time activ'
  KindOfSp   = 'Which is your major sporting activity?'
  Hours      = 'On average, how many hours a week do you'
  WhySport   = '@GWhy do you engage in @W^KindOfSport?'
  AnySport   = 'Do you engage in any sport?'
  WhyNotSp   = 'Why do you not engage in sport?'
  AverageH   = 'AverageHours'
  TellWhy    = 'Why do you choose to do sport?'
  DateOfBi   = 'What is your birthday? @/@/@Y[INTERVIEWE'
  LunchTim   = 'What time do you take lunch?'
  NumberOf   = 'Number of members in the household.'
  Name01     = 'Name of household member.'
  Name02     = 'Name of household member.'
  …
  Name10     = 'Name of household member.'
;

FORMAT
  FreeTim1  TE_1F.
  FreeTim2  TE_1F.
  …
  FreeTim6  TE_1F.
  KindOfSp  TE_3F.
  WhySport  TE_2F.
  AnySport  TE_4F.
;

RUN;
```

To use this SAS setup:

- Export the Blaise data as an ASCII file with the name `activity.asc`.
- Run this setup (`activity.sas`) in SAS, with the `activity.asc` file as the input text file.

9.6 Programming in Cameleon

At times, the data descriptions produced by the standard Cameleon translators may not be sufficient for your needs. New software may appear, or you may have project-specific requirements that are not covered by the existing software. In these cases, you might need to create your own Cameleon translators.

9.6.1 Basic Cameleon programming concepts

When you create a Cameleon translator program, certain basic information is crucial:

- By default, Cameleon writes all output to the file `cameleon.out`. However, you can change the name of that file by assigning another value to the predefined variable OUTFILE.
- Any text within square brackets [ ] is a program instruction, and text outside the brackets is treated as text to be written to the output file. This is a very important point, and it is *crucial* to understand this distinction. We will illustrate this in the example to follow, `cameltst.cif`.
- A bracket followed by an asterisk [*] denotes a comment and is not part of the program.

The *Reference Manual* contains a complete list of Cameleon reserved words and syntax.

9.6.2 Example program cameltst.cif

For example, consider the following program (`cameltst.cif`):

```
[*CAMELTST.CIF]
DATAMODELNAME                     [*Line 1]
[DATAMODELNAME]                   [*Line 2]
OUTFILE                           [*Line 3]
[OUTFILE]                         [*Line 4]
OUTFILE := 'NEW.OUT'              [*Line 5]
[OUTFILE := 'NEW.OUT']            [*Line 6]
METAINFOFILENAME                  [*Line 7]
[METAINFOFILENAME]                [*Line 8]
[COPY(METAINFOFILENAME,1,4)]      [* Line 9]
['COPY(METAINFOFILENAME,1,4) = ' + _
    COPY(METAINFOFILENAME,1,4)]   [*Line 10]
OUTFILE                           [*Line 11]
[OUTFILE]                         [*Line 12]
```

The program above gives the following output in `cameleon.out` for data model `leisure`, with the accompanying file `activity.bmi`:

```
DATAMODELNAME
Leisure
OUTFILE
CAMELEON.OUT
OUTFILE := 'NEW.OUT'
```

The following output is in `new.out`:

```
METAINFOFILENAME
Activity
Acti
COPY(METAINFOFILENAME,1,4) = Acti
OUTFILE
NEW.OUT
```

The following steps explain how Cameleon executed the lines of the program. Each item in the following list can be identified in the `cameltst.cif` file itself by the line labels *[*Line 1], [*Line 2], [*Line 3],* and so on:

[*Line 1]: The string DATAMODELNAME is written to the default output file (`cameleon.out`) because it is not enclosed in [ ].

[*Line 2]: The value of the Cameleon data model variable called DATAMODELNAME is written to the default output file (`cameleon.out`) because DATAMODELNAME is enclosed in [ ]. In this case, the value is *Leisure.*

[*Line 3]: The string OUTFILE is written to the default output file (`cameleon.out`).

[*Line 4]: The value of the Cameleon output file (`cameleon.out`) is written to the default output file.

[*Line 5]: The string OUTFILE := 'NEW.OUT' is written to the default output file.

[*Line 6]: The default output file is redefined to `new.out`.

[*Line 7]: The string METAINFOFILENAME is written to the output file (`new.out`).

[*Line 8]: The value of the Cameleon data model variable called METAINFOFILENAME is written to the output file (`new.out`). In this case, the value is `Activity`.

[*Line 9]: The first four characters of the value of the Cameleon data model variable called METAINFOFILENAME are written to the output file (`new.out`). In this case, the value is *Acti.*

[*Line 10]: The string COPY (METAINFOFILENAME,1,4) = is written to the output file (`new.out`), plus the first four characters of the value of the Cameleon data model variable called METAINFOFILENAME. In this case, the output is COPY(METAINFOFILENAME,1,4) = ACTI.

[*Line 11]: The string OUTFILE is written to the output file (`new.out`).

[*Line 12]: The value of the Cameleon output file (`new.out`) is written to the output file (`new.out`).

9.6.3 Example program param.cif

In Section 9.4.2 we saw that one of the options available was a field *Parameter*, and in that field it is possible to enter parameters used by the Cameleon translator. The translator `param.cif` uses those parameters or command line parameters:

```
[*PARAM.CIF]
['Data model name = ' + DataModelName]
['Metadata file name = ' + MetaInfoFileName]
['First parameter = ' + Parameter(1)]
['Second parameter = ' + Parameter(2)]
```

And if we use the same file and data model together with the parameter string *One;two,* then the output in `cameleon.out` is:

```
Data model name = Leisure
Metadata file name = Activity
First parameter = One
Second parameter = Two
```

Note that the two parameters were separated by a semicolon (;).

9.6.4 Example program wesvar.cif

WesVar is software for computing estimates and their standard errors when data are collected using a complex sample. The software was developed by Westat, and a 30-day trial version is available free of charge from the Westat Web site. For further information on that software including pricing information, see the Westat Web site.

The `wesvar.cif` file contains a Cameleon translator to produce data descriptions of the data for a Blaise data model, where the data are output in a

rectangular ASCII file by Manipula. The `wesvar.cif` program is not complex and helps us understand some basic Cameleon features. `Wesvar.cif` is contained in the following examples:

```
[*WesVar.CIF]
[** Variable declarations for the whole translator file]
[VAR
     FLDUNIQUE          : STRING,
     STARTPOSN          : STRING,
     FLDWIDTH           : STRING,
     NextPosn           : REAL]
[** Setting some environment variables]
[INDENT:= TRUE]
[LAYOUT:= TRUE]
[MAXNAMELENGTH := 11]
[NextPosn := 1]
[OUTFILE := COPY(DATAMODELNAME,1,8)+'.DAT']
[** Variable declarations specially for the procedure]
[VAR
     OrigStr            : STRING,
     NewStr             : STRING,
     OrigLnth           : REAL,
     CharToken          : STRING,
     AtToken            : STRING,
     HasAt              : BOOLEAN,
     AtPosn             : REAL,
     I                  : REAL]
[** Procedure checks for @, so set as variable]
[AtToken := '@']
```

```
[PROCEDURE StripAt]
  [NewStr := '']
  [HasAt := FALSE]
  [OrigLnth := LENGTH(OrigStr)]
  [** Step through original string]
  [FOR I :=1 TO OrigLnth DO]
    [CharToken := COPY(OrigStr,I,1)]
    [** Check for @]
    [IF CharToken = AtToken THEN]
      [HasAt := TRUE]
    [ELSE]
      [** Check to see if last chracter was @]
      [IF NOT HasAT THEN]
        [** Copy chracter to new string]
        [NewStr := NewStr + CharToken]
      [ENDIF]
      [HasAt := FALSE]
    [ENDIF]
  [ENDDO]
[ENDPROCEDURE]

[** Main program:]
[** Steps through blocks, fields, arrays, and sets]
[BLOCKPROC]
  [FIELDSLOOP]
    [ARRAYLOOP]
      [IF TYPE <> BLOCK THEN]
        [IF TYPE <> OPEN THEN]
          [SETLOOP]
            [FLDUNIQUE := UNIQUENAME]
            [WHILE LENGTH(FLDUNIQUE) < 11 DO]
              [FLDUNIQUE := FLDUNIQUE + ' ']
            [ENDDO]
            [STARTPOSN := STR(NextPosn)]
            [WHILE LENGTH(STARTPOSN) < 5 DO]
              [STARTPOSN := ' ' + STARTPOSN]
            [ENDDO]
            [FLDWIDTH := STR(FIELDLENGTH)]
            [WHILE LENGTH(FLDWIDTH) < 3 DO]
              [FLDWIDTH := ' ' + FLDWIDTH]
            [ENDDO]
            [FLDUNIQUE + ' ' + STARTPOSN + ' '+ _
                FLDWIDTH] [&]
            [OrigStr := FIELDLABEL]
            [StripAt]
            [NextPosn := NextPosn + FIELDLENGTH]
            [' ' + copy(NewStr,1,16)]
          [ENDSETLOOP]
        [ENDIF]
      [ELSE]
        [BLOCKCALL]
      [ENDIF]
    [ENDARRAYLOOP]
  [ENDFIELDSLOOP]
[ENDBLOCKPROC]
```

Wesvar.cif basic elements

`Wesvar.cif` has the following basic elements:

- Variable declarations for the main routine ([VAR … NEXTPOSN : REAL]).
- Variable declarations that will be used only by the procedure *StripAt*. That is, [VAR … I : REAL].
- Assignment of a value to a constant for this procedure ([ATTOKEN := '@']).
- Definition of a procedure ([PROCEDURE STRIPAT] … [ENDPROCEDURE]) that strips out the formatting symbols from field descriptions (or question text). Note that Cameleon procedures do not take parameters and all Cameleon variables have global scope. In this case, the global input variable is *OrigStr* and the global output variable is *NewStr*. The procedure uses a FOR-DO … ENDDO loop.
- A main routine that steps through each data field in the data model, collecting information ([BLOCKPROC] … [STRIPAT] … [ENDBLOCKPROC]).

Wesvar.cif: example output

We will now look at the output from `wesvar.cif` for data model `leisure`.

In the case of this output, notice that:

- The names of the WesVarPC variables are the same as those for both SPSS and SAS (for example, NAME01 … NAME10). These variable names are generated by the Cameleon variable UNIQUENAME.
- The variable label for *WhySport* is '*@GWhy do you …*' for both SPSS and SAS; in the case of WesVar, the label is *Why do you engag*. That is, for WesVar, the formatting symbols *@G* have been removed from the text. The procedure *StripAt* removes the formatting symbols from text after the text is assigned to the variable *OrigStr*.

```
FreeTime1        1   1 What is your fav
FreeTime2        2   1 What is your fav
…
FreeTime6        6   1 What is your fav
KindOfSport      7   1 Which is your ma
Hours            8   3 On average, how
WhySport        11   1 Why do you engag
AnySport        12   1 Do you engage in
WhyNotSport     13  20 Why do you not e
AverageHour     33   5 AverageHours
DateOfBirth     38   8 What is your bir
LunchTime       46   8 What time do you
NumberOfMem     54   2 Number of member
Name01          56  20 Name of househol
Name02          76  20 Name of househol
…
Name10         336  20 Name of househol
```

9.6.5 Analysing the wesvar.cif translator

In the procedure, each successive character of *OrigStr* is examined (*CharToken*) and compared to @ *(AtToken)*. If the character is not @, then the character is copied to *NewStr*. Otherwise, two characters are skipped and not copied to *NewStr*.

```
 [PROCEDURE StripAt]
  [NewStr := '']
  [HasAt := FALSE]
  [OrigLnth := LENGTH(OrigStr)]
  [FOR I :=1 TO OrigLnth DO]
    [CharToken := COPY(OrigStr,I,1)]
    [IF CharToken = AtToken THEN]
      [HasAt := TRUE]
    [ELSE]
      [IF NOT HasAT THEN]
        [NewStr := NewStr + CharToken]
      [ENDIF]
      [HasAt := FALSE]
    [ENDIF]
  [ENDDO]
[ENDPROCEDURE]
```

In the main part of the translator code, there is a block procedure ([BLOCKPROC] … [BLOCKCALL] … [ENDBLOCKPROC]), and all instructions within that block procedure relate to the 'current' block in the data model. What constitutes the current block is defined by other instructions, but the data model itself is the first block. Remember that a block contains fields, and some fields may be blocks.

```
[BLOCKPROC]
  [FIELDSLOOP]
    [ARRAYLOOP]
      [IF TYPE <> BLOCK THEN]
        [IF TYPE <> OPEN THEN]
          [SETLOOP]
            [FLDUNIQUE := UNIQUENAME]
            [WHILE LENGTH(FLDUNIQUE) < 11 DO]
              [FLDUNIQUE := FLDUNIQUE + ' ']
            [ENDDO]
            [STARTPOSN := STR(NextPosn)]
            [WHILE LENGTH(STARTPOSN) < 5 DO]
              [STARTPOSN := ' ' + STARTPOSN]
            [ENDDO]
            [FLDWIDTH := STR(FIELDLENGTH)]
            [WHILE LENGTH(FLDWIDTH) < 3 DO]
              [FLDWIDTH := ' ' + FLDWIDTH]
            [ENDDO]
            [FLDUNIQUE + ' ' + STARTPOSN + ' '+ _
                FLDWIDTH] [&]
            [OrigStr := FIELDLABEL]
            [StripAt]
            [NextPosn := NextPosn + FIELDLENGTH]
            [' ' + copy(NewStr,1,16)]
          [ENDSETLOOP]
        [ENDIF]
      [ELSE]
        [BLOCKCALL]
      [ENDIF]
    [ENDARRAYLOOP]
  [ENDFIELDSLOOP]
[ENDBLOCKPROC]
```

In the block procedure, there is a loop through all the fields in that block ([FIELDSLOOP] … [ENDFIELDSLOOP]), and if the field itself is a block, then the block procedure is called again ([IF TYPE <> BLOCK THEN] … [ELSE] [BLOCKCALL] [ENDIF]). Most of the coding is concerned with formatted output. Note that everything sent to the output file (in this case, [OUTFILE := COPY(DATAMODELNAME,1,8) + '.DAT']) has to be a number or text. If more than one item is output at the same time (for example, [FLDUNIQUE + ' ' + STARTPOSN + ' ' + FLDWIDTH]), each item has be text or a string variable.

9.6.6 Using metadata loops

Cameleon has two conventional iterative looping structures: WHILE…ENDDO and FOR…ENDO. In addition, Cameleon has *metadata loops,* the use of which can be illustrated by the basic structure of the block procedure:

```
[BLOCKPROC]
  [FIELDSLOOP]
    [ARRAYLOOP]
      [IF TYPE <> BLOCK THEN]
        [IF TYPE <> OPEN THEN]
          [SETLOOP]
            …
          [ENDSETLOOP]
        [ENDIF]
      [ELSE]
        [BLOCKCALL]
      [ENDIF]
    [ENDARRAYLOOP]
  [ENDFIELDSLOOP]
[ENDBLOCKPROC]
```

That is, there are metadata loops within metadata loops:

- There is a metadata loop through all fields (FIELDSLOOP … ENDFIELDSLOOP).
- For each field there is a metadata loop (ARRAYLOOP … ENDARRAYLOOP) through all instances of that array (or only once, if the field is not an array).
- If the field is an instance of a block, then move to the ELSE condition for that check.
- If the field is an instance of an OPEN type of question, then move to the ENDIF condition for that check.
- For each element of the array for that field there is a metadata loop (SETLOOP … ENDSETLOOP) through all instances of that set (or only once, if the field is not a set).

In `sas.cif` there are further metadata loops:

```
[BLOCKPROCEDURE]
    [TYPESLOOP]
    [IF (TYPE = BLOCK) AND NOT PREDEFINEDTYPE THEN]
      [BLOCKCALL]
    [ELSEIF (TYPE = ENUMERATED) OR ((TYPE = SET) AND _
        (DEFINEDTYPENAME = '')) THEN]
      []VALUE TE_[ENUMTYPENUMBER]F
      [ANSWERLOOP]
        [IF ANSWERTEXT = '' THEN] [:2][ANSWERCODE]='[ANSWERNAME]'
        [ELSE] [:2] [ANSWERCODE]='[ANSWERTEXT]'
        [ENDIF]
      [ENDANSWERLOOP]
      [];

    [ENDIF]
  [ENDTYPESLOOP]
[ENDBLOCKPROC]
```

In this case, there are two loops within a block procedure:

- A *types loop* (TYPESLOOP) that loops through all the types in the block.
- An *answers loop* (ANSWERLOOP) that loops through all the answer codes for an enumerated type.

The output for this section is:

```
VALUE TE_1F
  1='Playing sport'
  2='Watching TV'
  3='Eating out'
  4='Going to pub or bar'
  5='Sleeping'
  6='Reading'
;

VALUE TE_2F
  1='Competition'
  2='Relaxation'
  3='Health benefits'
  4='Weight loss'
  5='Other'
;

VALUE TE_3F
  1='Soccer'
  2='Cycling'
  3='Tennis'
  4='Aerobics'
  5='Bridge'
  6='Chess'
  7='Other'
;

VALUE TE_4F
  1='Yes'
  2='No'
;
```

Note that VALUE TE_4F was not defined in the TYPE section of the data model, and in Blaise all enumerated types have a distinct number (ENUMTYPENUMBER) whether they have been formally defined or are part of a FIELDS declaration.

10 CATI Call Management System

Blaise® supports Computer Assisted Telephone Interviewing (CATI). Blaise® CATI is easy to set up and use, is parameter driven, is compatible with Computer Assisted Personal Interviewing (CAPI), and needs little if any special hardware.

Blaise® uses the power of networked computers to accomplish several things necessary for an effective CATI survey, including:

- Shared data files
- Joint handling of busy signals, so that the number will be tried again in a few minutes
- Management of appointments for individual interviewers and groups of interviewers, so that forms can be routed back to an individual or group member who established rapport
- Special strategies for handling no-answer or answering machine calls with time slices. For example, if a call is made during an afternoon and no one is reached, you might want to call in the evening instead
- Up to the minute survey management
- Tracking and enforcing quotas
- Time zones
- Supervisory intervention
- Reports

Several parties are involved in a CATI survey and each one has a different perspective on the CATI system.

The interviewer conducts the interview and uses CATI survey management features such as the dial menu and the appointment dialog box when conducting the survey in the Data Entry Program (DEP).

The developer includes the correct CATI instructions in the data model, such as the INHERIT CATI setting, the special telephone, route-back, to whom, and time zone fields. The developer also sets up the parallel blocks that interviewers will need to make appointments and record other call results.

The study manager uses the CATI Specification Program to set survey parameters such as how forms are delivered, valid survey days, and interviewer names.

The CATI supervisor uses the CATI Management Program to monitor calls and to intervene as needed.

It is important to coordinate all these people so that you can increase the response rate, implement special sampling plans, and reduce costs by increasing productivity.

The Comtel example used in this chapter is a person-level instrument about commuting with known respondents and telephone numbers.

10.1 Blaise CATI Concepts

There are several basic concepts in a Blaise CATI survey. In this section we discuss those concepts. Chapter 11 provides additional technical details about the Blaise CATI system and includes a glossary of CATI terminology.

CATI instrument

The first step is to create an instrument that contains the special settings and blocks needed for a CATI survey. Details on developing data models are in Chapters 3, 4, and 5. The steps necessary for a CATI model are covered later in this chapter.

Blaise data file

Before the survey starts you initialise or pre-load a Blaise data file with data. At a minimum, a telephone number is required, but you can also load other administrative information, such as name and address, time zones, or previously collected data.

When a form is delivered to a workstation, the data in that form (because it was initialised with data) are copied from the server to workstation memory. When the interviewer completes that form, the data are copied to the server. In this way, the data file on the server is updated continuously with results from all the interviewers.

CATI specification

You create a CATI specification file for your survey using the CATI Specification Program (select *Tools ➢ CATI Specification* from the menu). The specification file contains parameters or settings that describe when and how a survey should be conducted, such as the survey period, the days on which interviews will be held, and the number of interviewers per shift. It also indicates which treatment must be given to telephone numbers in various situations.

The Blaise data file that you initialised may be very large. Manipulating the entire file at one time could cause problems with the system's performance. Blaise solves this problem by using a *daybatch*. A daybatch is a file that contains forms for only those respondents who may be contacted on a specific day in the survey period.

The daybatch is created before interviewing begins. You influence the composition of the daybatch through items defined in the CATI Specification Program, but create it manually or in batch overnight using the CATI Management Program.

Workstation and interviewer

To carry out the survey, interviewers run the Data Entry Program (DEP). By default, each time the interviewer asks for a new form, a dial screen appears with the number to be contacted. The interviewer either conducts the interview or records a different call outcome, such as making an appointment or recording a refusal.

Appointments are an important part of the system. Interviewers can make an appointment at any time.

Dial and dial result

A dial is the process of selecting a form and making it available to the interviewer to try to get a respondent on the line. In other words a dial is an attempt to contact the survey subject. After conclusion of the dial, the form acquires a dial result. A dial result determines what treatment should be given to a form.

The eight high-level dial results supported by Blaise are:

- *Response*: the questionnaire is complete.
- *No answer*: no one answered after several rings.
- *Busy*: a regular busy signal (not a fast busy) is heard.

- *Appointment*: An appointment is recorded for a later contact.
- *Non-response*: The respondent refuses to cooperate.
- *Answering service*: An answering machine or service was reached. The interviewer may or may not have left a message.
- *Disconnected*: The phone number was disconnected.
- *Others*: Other situations can be recorded.

For each dial result, it is possible to include questions in a parallel block that prompts the interviewer for additional detail. For example, if a refusal is recorded, it may be firm or mild and this may help determine if the respondent should be contacted again at a later date. The treatment a form gets depends on the dial result (see the following section).

Call

A call is a logically related set of dials. This means that a call is made up of one or more dials depending on the history of the form. Some examples:

- Dial result is *Response*: One dial and one call.
- Dial results are *No answer*, *No answer*, then the form goes to *No need today* (because *Maximum number of dials = 2)*. There are 2 dials and 1 call.
- Dial result is *No answer* and the calling day ends before there is another opportunity to attempt this form: One dial and one call.

Since dials are subsets of calls, there may be many more dials for a case than the number of calls. The history file records dial results (all attempts). The CATI Specification File setting *General parameters/Day batch parameters/Maximum number of calls* refers to the number of calls as defined here (not the number of dials or attempts).

Treatment

A treatment covers the process by which the scheduler determines the appropriate status code and other items that govern when the form should next be called or next be eligible for daybatch creation. Treatments have several aspects that can be controlled by the datamodel developer or through CATI specifications. Thus you have the ability to determine or influence how each form is handled depending on its treatment and the history of the form itself. Treatments apply to several non-concluding dial results.

The CATI system distinguishes a number of non-concluding treatment types:

- A no-answer dial result gets the treatment *no-answer.*
- A busy dial result gets the treatment *busy.*
- An appointment dial result gets the treatment *appointment.*
- An answering service dial result gets the treatment *answering service*.

The term *treatment* can be considered very broadly. It may include a parallel block with additional questions attached to the dial result, the values of parameters in the specification file, datamodel rules to handle certain situations, and other actions that determine what to do with each form.

Forms with the dial results *response*, *disconnected*, *nonresponse,* and *other* are considered concluded, and therefore no treatment is needed. It is possible to bring these forms back into the scheduler on a case-by-case basis or en masse for a class of forms with other Blaise management tools such as Manipula or Maniplus, but this is beyond the scope of this chapter. For example, a form with an *other* dial result may represent a situation where there was a language problem. This form can be refielded if an interviewer can be found that can speak this language.

Call scheduler

Scheduling is the method by which forms in the daybatch are made available to be dialled. The part of the program that performs this function is called the scheduler. The scheduler assesses the priority and the status of the forms in the daybatch and adjusts them if necessary. It takes into account appointments, busy signals, and no-answer call backs and selects the proper treatment for unsuccessful call attempts.

The scheduler works in five-minute intervals. The first interviewer asking for a new number in an interval causes the system to update the priorities. New priorities are assigned to active numbers in the daybatch, and this process is called *scheduling*. Although the daybatch normally contains the same numbers all day, only a subset of the numbers is active at any one time, and this subset varies continuously. The priority assigned to a form may change many times.

The daybatch is updated by the workstation of the interviewer who invokes the update. An extra workstation is not necessary to run the CATI scheduler.

Statuses in the daybatch and Priorities

The scheduler works by assigning one of the following statuses to forms in the daybatch:

- *Being treated:* A dial is currently being performed for this number.
- *Busy:* The result of the last dial was busy, and the form will be given the busy treatment.
- *No-answer:* The result of the last dial was no-answer or answering service, and the form will be given the no-answer treatment.
- *Appointment:* An appointment for today has just been made for this form. The next schedule will determine whether the new status should be active or not-active.
- *No need today:* This form should not get another dial in the current daybatch.
- *Not-active:* The form cannot get a dial at the moment. The form will be given the status *active* at a certain time.
- *Active:* The form is available for a dial.

Statuses are assigned based on the previous dial result, specifications in the CATI specification file, the date and time, which interviewers and groups are on the system at the moment, and quota information.

In the CATI Management Program, under the *View Daybatch/Browse* branch the *Status/Priority* column displays the status of the form unless the status is *Active*. If the status is *Active* then the priority of the form is displayed in this column. Priorities are *Super*, *Hard-busy*, *Hard*, *Medium-busy*, *Soft-busy*, *Default-busy*, *Medium*, *Soft*, and *Default*. Priorities are fully documented in Figure 11-1 of chapter 11.

Time slices

Time slices are a feature of the no-answer treatment, which help you handle no-answer call backs. If you have dialled a number during a certain time of the day and continue to get a no-answer response, you might want to try a different time of the day to dial that number. You can create *time slices* that divide the survey day into different time periods. Then, based on settings in the CATI specification file, the scheduler will keep track of how many dials were made in that time slice and schedule that form to be called back during a different time slice.

CATI Management Program

A supervisor manages the survey using the CATI Management Program. It is from within this program that the supervisor monitors progress, browses the forms in the daybatch, applies treatments to forms, and views active interviewers and groups.

Counts file

The system creates a file that stores the number of completes, no-answers, and other outcomes of dial attempts. These counts are shown in the *Summary* and *Quota* branches of the CATI Management Program.

Quota

If there is a quota to fill based on the value of a field in the data model, and then the counts file keeps track of the number of responses of this value for all interviewers.

Quotas can be implemented at two levels: at the time of form delivery or in the RULES section of the data model. You can include a condition on whether a quota is reached or not. For example, if the quota is reached, then you can reduce respondent burden by skipping past parts of the questionnaire.

History file

As dial attempts are made, the results of the attempts are kept in the history file. This file can be inspected within the CATI management program or by using the `bthist.exe` program that is in the Blaise system folder.

Log file

The log file keeps track of system-level events such as when a daybatch is created, when interviewers log on and off, and so on. It is an ASCII file with a `.log` extension that can be viewed in any file browser.

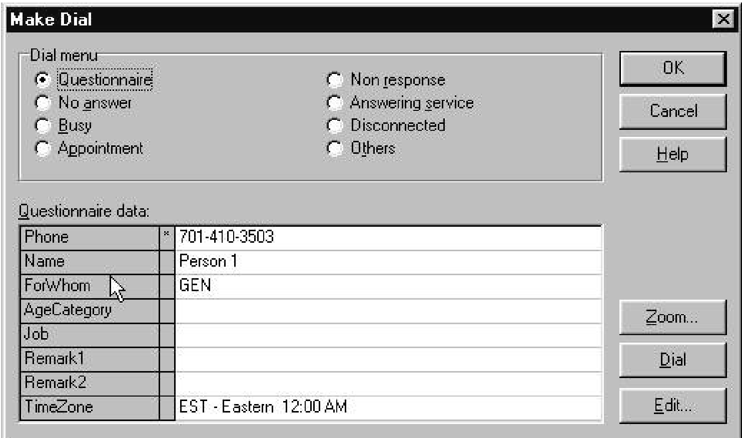
10.2 CATI Interviewing

Using the Data Entry Program (DEP) for CATI is very similar to using it for other data models with a few exceptions. You must use a CATI menu and optionally, CATI surveys can use a *Make Dial* screen and the default appointment-making dialog.

10.2.1 Make Dial screen

By default, when an interviewer requests a form, the optional *Make Dial* screen is the first screen the interviewer sees. The dial screen can be disabled by setting the *Skip dial* menu in the *DEP* option on the *dial* menu dialog in the CATI specification program. The following sample shows a typical dial screen:

Figure 10-1: Dial screen



The *Dial menu* and *Questionnaire data* sections are customisable in the CATI specification file. The *Dial menu* section requires the *Questionnaire* button and the *Questionnaire* section requires the phone number field.

In this example, all 8 dial results are displayed in *Dial menu*. Another common configuration is to display only the *Questionnaire* button and require the interviewer to get into the instrument before calling the respondent. This allows the interviewer to follow a scripted contact procedure.

In the *Questionnaire data* section, it is common to display any information the interviewer should know before contacting the respondent, including form notes from the previous dial attempt.

Dial menu section

In the *Dial menu* section the *Questionnaire* radio button is required but the display of the other seven dial results is optional (they can still be accessed from within the instrument). Typically if all eight options were displayed, the interviewer would dial the number, speak to the respondent, and then select an

option from the *Dial menu* section. If a modem were available to dial, the interviewer would click the *Dial* button to dial the number.

To start the interview, select the option that starts the interview. On the example above, it is *Questionnaire*, the first option listed. The exact text that appears here is determined by the CATI specification file. The DEP runs the interview.

If the dial menu contains an *Appointment* option, you can make an appointment from the dial screen without starting the questionnaire. You can also make an appointment from within the questionnaire by selecting the Appointment parallel block. See the next section for more details on making appointments.

Some of the dial options may have parallel blocks that display when selected. For example, you might want some additional information if you encounter a nonresponse during interviewing. When that option is selected, the DEP displays the appropriate parallel where you can enter the additional information.

Questionnaire data section

In the *Questionnaire data* section only the phone number display is required. Here you can display any instrument information, such as form-level interviewer notes, that can help the next interviewer attempting an interview. You choose fields to be displayed in this section in the CATI specification file *Field selection* branch. The telephone number is the only field that is required to be displayed. The other fields provide the interviewer with information that might be helpful when making calls.

You may be able to edit (change) certain fields on the *Make Dial* screen itself. This is determined by a setting in the CATI specification file in the *Field selection* branch. If a field can be edited, an asterisk * appears in the column next to the field in the *Questionnaire data* section. In the example above, the only field that can be edited is *Phone.* The * changes to a + after a field value is edited (changed).

Zoom

To see more information on a specific form, click the *Zoom* button to access the *Case summary* box.

10.2.2 Making appointments

You can make appointments either from the dial screen (if allowed) or from within the interview. You can only make appointments for valid survey days.

From the dial screen, select the *Appointment* option. If you are in the interview and need to stop it and make an appointment, click on the tab for the appointment parallel block. If there is no appointment tab, select *Navigate ➤ Sub Forms* from the menu (or select the appropriate shortcut key) and select the *Appointment* option from the *Parallel Blocks* dialog box. The *Make Appointment* dialog box appears.

Figure 10-2: Make appointment dialog box showing a Weekday period appointment

There are several combinations of date and time that you can select, and the options available for time are dependent on the option selected for the date. The following table summarises the available combinations:

Figure 10-3: Time and date combinations when making appointments

	No Date	Exact Date	Period	Weekday
No time			*	*
Day part	*	*	*	*
Exact time		*		

Make your date and time selection as described in the following section, then click the *OK* button.

! You can use the mouse or the keyboard keys to move around in the *Make Appointment* dialog box. Press the Tab key to move from section to section. Use the up and down arrows to move around within the calendar or to select options from boxes. Use the left and right arrow keys to move within the time boxes.

You can disable Blaise's default appointment dialog in the *Parallel blocks* setting of the CATI Specification Program (see the following section).

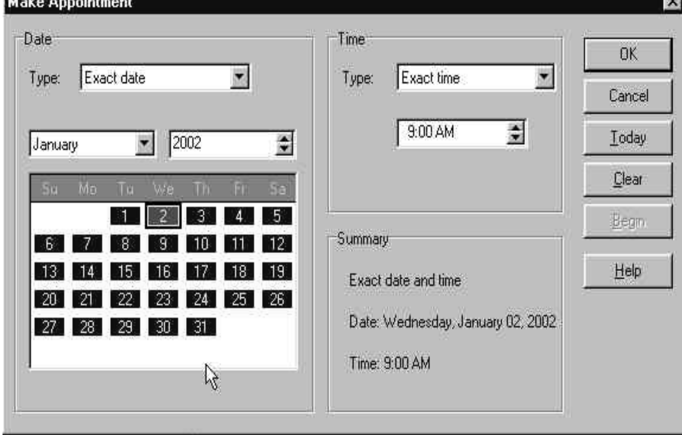
No specific date

To set an appointment for *no date*, just select that option. When choosing no date, the time choice is *day part* or *no time.* If you set an appointment for *no date* and *no time*, you create a *no preference* appointment.

Exact date

To set an appointment for an exact date, a calendar appears in the *Date* section. Valid survey days are highlighted on the calendar.

Figure 10-4: Setting an exact date appointment



Select a specific month and day from the boxes. You can also click once in the calendar, and use the page up and down keys to scroll through the months.

To select a date, double click on the date (or use the space bar to toggle). When choosing an exact date, you have a time choice of *exact time* or *day part*.

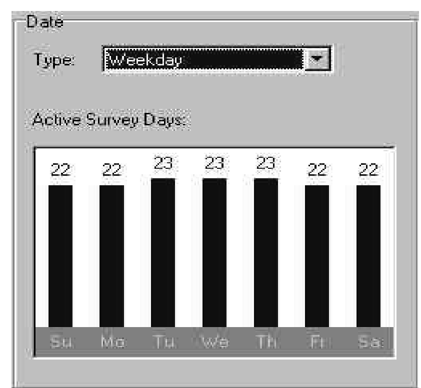
Period appointment

To set an appointment for a *period,* the calendar also appears. Select a range from the active survey days by double-clicking a begin and end date. When choosing a period, you have a time choice of *day part* or *no time.*

Weekday appointment

When setting an appointment for a *weekday*, a bar graph of the days of the week appears. Select one or more days of the week by double-clicking the appropriate bars. The number of available days for each weekday is at the top of each bar. When selecting a weekday, you have a time choice of *day part* or *no time.*

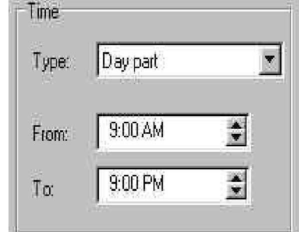
Figure 10-5: Setting a weekday appointment



Selecting an appointment time

You can select a *day part* or an *exact time* in the *Time* section. For a day part, select a beginning and ending time for a range; for an exact time, choose a specific time. The time increments available reflect the appointment interval setting in the CATI specification file.

Figure 10-6: Setting an appointment time



10.2.3 Using a CATI menu

A CATI menu file is necessary to run a CATI survey because you need to have certain options available. For example, the menu option *Forms* ➢ *Get Telephone Number* is another way to access the dial screen. If this menu option is not made available in the menu file, then the dial screen won't be available either.

There are times when you want to have additional methods of accessing forms, instead of going through the dial screen or the *Get Telephone Number* menu. Consider providing *Get Telephone Number*, *Browse, Get,* and *Exit* from the *Forms* menu.

Get Telephone Number

The *Get Telephone Number* option allows interviewers to re-invoke the automatic delivery of forms after they have used *Browse* or *Get* to access a particular case. When an interviewer first gets into the CATI system, this option is assumed and the *Make Dial* screen automatically appears.

Browse

The *Browse* option allows interviewers to browse through all the forms in the data file, not just the daybatch. For example, there may be more than one form for which a respondent has answers and you want one interviewer to handle all of them in succession.

You can set the *Browse* option to browse by primary or secondary key. A very useful secondary key is the telephone number.

Get

The *Get* option allows the interviewer to request a form by a primary key.

Using *Get* or *Browse* to access a form has no effect on how the dial screen or the form is handled by Blaise. The system handles the form just as if it were selected automatically. If the form is not part of the daybatch when accessed, it is added to the daybatch automatically.

Exit

You should provide a *Forms* ➢ *Exit* menu option to allow interviewers to quit CATI. When interviewers go on a break they should exit CATI so that the treatment time column in the History viewer is not artificially inflated.

Default CATI menu

There is a default CATI menu called `catimenu.bwm` in the Blaise system folder. You can use this menu as it is or edit and rename it to suit your needs. Information on creating and editing menu files can be found in Chapter 6.

10.3 Developing CATI Data Models

In most respects data models for CATI do not differ from data models for other types of surveys. But because CATI uses a call management function, you have to include special information in a CATI data model.

At a minimum, you need pre-coded CATI management blocks and a telephone number field in your data model. Optionally, you usually include other fields or blocks in the data model to optimise the use of the system.

In this section we will describe what you must do to adapt a data model for a CATI survey.

10.3.1 INHERIT CATI and TCatiMana

To make a data model suitable for use in a CATI survey, you must include a special block that contains fields that store management information.

INHERIT CATI

The special block is included with the following setting:

```
INHERIT CATI
```

This must be the last statement in the SETTINGS section of the data model, appearing just before the first FIELDS or INCLUDE statements.

INHERIT CATI does more than include pre-coded blocks. It also signals to Blaise that the data model will use the CATI functions of Blaise.

TCatiMana block

The included block is called *TCatiMana* and it contains three sub-blocks: *TAppMana*, *TCallMana*, and *TSliceMana*.

- *TAppMana* records information for appointments made by interviewers. This block has access to the information in the survey definition, such as the valid survey days. It knows when interviewing will take place and will not accept appointments that cannot be met.
- *TCallMana* stores information about the very first call and the last four calls. A call consists of one or more dials, depending on the dial result as documented above. Normally the call scheduler sets this information automatically. However, a custom call result from within the data model can be recorded in *TCallMana* by assigning a value to the auxfield *CallResult*. The result assigned using the *CallResult* auxfield overrides the result selected from the dial screen or the result from a parallel block.
- *TSliceMana* stores information for up to 32 dials. For each dial the system stores the weekday and the time of the dial. The dial time is registered in respondent time. The information will be used to determine which time slice definitions have to be blocked because enough dials have already been made during the slice.

A listing of the blocks is shown in the following code. You must not change these blocks. These are also in the file `cati.inc`, which is in the Blaise system folder.

```
{This file contains definitions of the fields that are used to store
data in the form about the CATI process}

 BLOCK TCatiMana "Cati Management";
 SETTINGS
   ATTRIBUTES = NoDK,NoRF

   TYPE
     TWeekDay = (Sunday, Monday, Tuesday, Wednesday,
                 Thursday,Friday,Saturday)

   BLOCK TAppMana "Appointment block for CATI";
   FIELDS
     AppointType "When can we call you back ?":
                  (NoPreference    "no preference",
                   CertainDate     "appointment for date and time",
                   Period          "preference for a period",
                   DayOfWeek       "preference for days of the week")
     DateStart   "start date       ": DATETYPE
     TimeStart   "start time       ": TIMETYPE
     DateEnd     "end date         ": DATETYPE
     TimeEnd     "end time         ": TIMETYPE
     WeekDays    "selected weekdays": SET OF TWeekDay
     WhoMade     "who made appoint ": STRING[10]
   ENDBLOCK {TAppMana}

   BLOCK TCallMana "Call management block for CATI"
   TYPE
     TCAllResult = (Completed "Questionnaire/Response",
                    NoAnswer      "No answer",
                    Busy          "busy",
                    Appointment   "Appointment",
                    NonResponse   "Non-Response",
                    AnswerService "Answering service",
                    DisConnected  "Disconnected",
                    Other         "No contact (rest)")

     BLOCK TCall

     FIELDS
       WhoMade "who made last dial in call": STRING[10]
       DayNumber "daynumber relative to FirstDay": 1..999
       DialTime   "time of last dial in call": TimeType
       NrOfDials  "number of dials in call  ": 0..9
       DialResult "result of last dial in call    ": TCallResult
     ENDBLOCK {TCall}

   FIELDS
     NrOfCall  "number of calls ": 1..99
     FirstDay  "date first call ": DATETYPE
     RegsCalls "registered calls": ARRAY[1..5] OF TCall
   AUXFIELDS
     CallResult: TCallResult
   ENDBLOCK {TCallMana}
```

```
  BLOCK TSliceMana
    BLOCK TDialData
    FIELDS
      WeekDay    "Weekday of slice dial" : TWeekDay
      DialTime   "Slice dial time"       : TIMETYPE
    ENDBLOCK {TDialData}

  FIELDS
    NrOfDials: 1..32
    DialData: ARRAY[1..32] of TDialData
  ENDBLOCK

FIELDS
  CatiAppoint    : TAppMana
  CatiCall       : TCallMana
  CatiSlices     : TSliceMana
RULES
  CatiAppoint.Keep
  CatiCall.Keep
  CatiSlices.Keep
ENDBLOCK {CatiMana}

FIELDS
  CatiMana: TCatiMana
```

You can access these fields in other parts of the data model or through Manipula or Maniplus.

10.3.2 Special CATI fields

A CATI survey requires some extra information to be stored in fields of your data model. At a minimum, you must include a telephone number field, but you might also include a route-back field, a time zone field, a time slice field, and one or more quota fields.

Define these fields as you would any other field in a data model, and then give them special status in the CATI Specification Program. The values of some of these fields are not obtained during the interview. You have to fill some of the fields during the initialisation step and the CATI Call Management System fills others.

Telephone field

Blaise expects the telephone number to be part of the data model. Therefore, you must have a field in the data model to store the telephone number. It is best to use a text field for this, because it usually contains non-numeric characters such as - *or* ( ). Be sure to make the field wide enough to store every possible telephone

number. If you use a modem, make sure the string of numbers can be interpreted by the modem software and includes all the digits necessary to access an outside telephone line.

You can use any field name. Later, you must assign a special meaning to this field in the CATI Specification Program.

Here is an example:

```
FIELDS
  Phone "Telephone number": STRING[30]
```

The telephone field must be filled in before you start the field work of your CATI survey. A convenient way to do this is to use Manipula to import an ASCII file with telephone numbers, and possibly other administrative information.

Route-back field and the To Whom function

The CATI Call Management System can route forms to a specific interviewer or group of interviewers by using a route-back field (also referred to as the *ToWhom* field or function). To use this option, include a special field for it in your data model. It should be a text field of sufficient length to store the interviewer or group identification. Here is an example:

```
FIELDS
  ToWhom : STRING[9]
```

The field name is arbitrary. If you do not care to which interviewer group a telephone number is routed, you do not have to include this field. If you fill the *ToWhom* field before starting a survey, you can make sure certain forms are given to specific interviewers or groups.

After the survey starts, the information can be updated by the CATI system when an appointment is made. Then the forms can be routed back to the interviewer who made the appointment or to an interviewer from the same group of interviewers. This feature is explained later in this chapter.

Routing forms back to interviewers only works in a network environment. Interviewers are identified by their login name, or alternatively, a registry entry for the key called *BlaiseUser* (in the environment subfolder of the HKEY_CURRENT_USER subfolder, you may have to create the subfolder and the key). The registry entry overrides the login name.

Time zone field

If your survey will be reaching respondents across time zones, you must have a field in your data model that contains the identification of the time zone. Time zones are stored as three-letter codes, so the time zone field must be a string field of three characters. In the CATI Specification Program, tell the system what the time differences for the various time zones are. An example of a time zone field is:

```
FIELDS
  TimeZone "Time zone": STRING[3]
```

You can use any field name. The time zone codes must be all uppercase. Time zone codes normally should be pre-loaded before you start the field work. An easy way to do this is to load these codes when you load the telephone numbers. Make sure the same codes are defined in the CATI Specification Program in the *Time Zone* settings.

Time slice field

You can divide the survey day into different divisions in order to schedule default-priority no answer call backs. These divisions are called “time slices,” and are defined in the CATI specification file. On a given day, the system will record the time slice a call was made in and schedule the call back for that form for a different time slice.

You can define several time slice sets. If the forms in your survey will use different time slice sets, you must include a time slice field in your data model that contains the identification of the time slice set. Time slice set identifiers are stored as three-letter codes, so the time slice field must be a string field of three characters. In the CATI Specification Program, you tell the system what the time slice set looks like.

An example of a time slice field is:

```
TimeSlice "Time slice set identifier":STRING[3]
```

You then provide the time slice information, including which time slice set codes to use for which forms.

If all your numbers use the same time slice set, you don't need to have a time slice field. The system will use the definition of the first time slice set by default.

Quota field

In quota sampling, the survey is designed to interview given quotas of particular groups of people. This requires a special field. Quota sampling might affect delivery of forms or it might affect how RULES sections within the data model are handled. For example, it can be used to ensure that sections of the questionnaire are asked or skipped based on a value of an enumerated field.

A quota field must be an enumerated field type. For example, suppose you had the field MarStat with enumeration labels (quota groups) *Married, Single, Divorced, Separated,* and *Widowed.*

You then set the specific maximum values for these quota groups : 100 for Married, 50 for Single, and 25 for Divorced (these values are set in the CATI specification file). At some point in the interviewing process, the marital status question is asked and the answer determined. You can then base routing or other instructions on the quota within the RULES section of the data model. For example:

```
IF NOT(QUOTAREACHED) THEN
  EmployerPolicy
  WorkPolicy(Whom, Employer, EmployerPolicy)
ENDIF
```

The system keeps track of all the responses from the interviewers. It knows when a quota has been reached and will execute the rules based on the quota. In the example above, the questions *EmployerPolicy* and *WorkPolicy* will be asked unless this form is the 26th with a value of *Divorced*, the 51st with a value of *Single*, or the 101st with a value of *Married*.

You can cross two or more fields to make one quota. The values are set in the CATI Specification Program. For example, you can make a field *AgeCategory* with the values *UpTo21, From21To40, From41To60*, and *Over60.*

These can be computed from the field *Age* which appears in the interview. These quota groups can be crossed with the *MarStat* quota groups and values of the crossing are set in the CATI specification file. If no quota is mentioned in a crossing, then there is no maximum number for the crossing.

10.3.3 Appointment block

Making appointments is an important part of a CATI survey. Appointment information is stored in the TAppMana block of the data model. If you want the ability to make appointments using the appointment dialog in the DEP, you must also include a second appointment block in the data model.

The fields in this block are used in CATI to store information that you want the interviewers to see in the *Questionnaire data* section of the dial screen. For example:

```
BLOCK BAppoint
  FIELDS
    Remark1  "@Y[INTERVIEWER] Make any general
              appointment remarks." : STRING[40], EMPTY
    Remark2  "@Y[INTERVIEWER] Room for more general
              appointment remarks." : STRING[40], EMPTY
ENDBLOCK
```

Note that you should not ask for a day and time for an appointment in this block. This is automatically taken care of by the system.

You can disable Blaise's default appointment dialog in the *Parallel blocks* settings of the CATI Specification Program. If you do this, you must define fields to record date and time information. See the *CATI Specification* section later in this chapter.

In the FIELDS section of your data model, you must introduce a field of the appointment block type. For example:

```
FIELDS
  Appoint: BAppoint
```

You should not include this block field name in the RULES section of the data model, or do so only under carefully chosen circumstances. You want to ensure that the questions in the block will not be asked during the normal course of the interview. If the instrument is to be used in both the CATI and interactive editing mode, and you want the data editor to see the appointment block without having to use the menu to access it, then you need to put it on the route in the RULES section. For example:

```
ThankYou
 IF (ThankYou=EMPTY) or CADI THEN
   Appoint
 ENDIF
```

In order to make the appointment block available from the *Navigate* menu of the Data Entry Program, you must make the appointment block a parallel block.

In the parallel setting, you can assign an identifying parallel name to the appointment block field. If this parallel name is *Appointment*, the system will assume that this block is to be treated as the appointment block. If you assign an arbitrary name to the block field, you must use the CATI Specification Program (*Parallel blocks* settings) to tell the system that this block field must be executed in the appointment situation.

10.3.4 Additional blocks

There are a number of other dial results for which you can specify special parallel blocks. You can include a parallel block for each treatment type. The following table lists parallel blocks for all possible dial results. The proper treatment for each block is set in the CATI specification file (described later in this chapter).

Figure 10-7: Parallel block names for dial results

Parallel Block Name	Dial Result
NOANSWER	There is no answer.
BUSY	The line is busy.
APPOINTMENT	The interviewer wants to make an appointment.
NONRESPONSE	The interviewer wants to record a nonresponse.
ANSWERINGSERVICE	The line is connected to an answering device or service.
DISCONNECT	The number is disconnected.
OTHERS	Other outcomes.

For example, if you want to ask some questions in case of a refusal to co-operate in the survey, you could include a nonresponse block like the following:

```
BLOCK BNonResp
  PARAMETERS
    IMPORT Whom : STRING

  FIELDS
    Reason " @Y[INTERVIEWER] Enter the reason for non-
            response for @B^Whom@B." : NonRespStatus
  RULES
    Reason
ENDBLOCK

FIELDS
  NonResp : BNonResp
```

To attach the nonresponse treatment to the block field *NonResp*, add the following statement to the parallel section:

```
PARALLEL
  Nonresponse = NonResp
```

Alternatively, if you name the nonresponse block as follows:

```
FIELDS
  NonResponse : BNonResp
```

you can state the parallel block as shown:

```
PARALLEL
  NonResponse
```

If you do not use either of these naming conventions, you can still give the block nonresponse treatment in the *Parallel blocks* settings in the CATI Specification Program.

10.3.5 Initialise the data file

Before the survey is started, all telephone number fields must be filled. You might also want to fill other administrative fields, such as names, addresses, time zones, or time slice set codes. One way to initialise the file with telephone numbers is to prepare an ASCII file with initialisation information and use Manipula to import the information into the Blaise data file.

Blaise is especially well suited for conducting surveys from a list of telephone numbers. The list may come from an administrative source, a sampling frame, or a third party supplier.

Blaise can handle list-assisted random digit dialling where random numbers are pre-generated and pre-screened for blocks of households. The way that telephone numbers are divided between residences, business, government, fax, and cellular telephone varies tremendously from country to country, and thus is out of the scope of this volume. However, a third party sampling business may exist and may be able to supply the list of telephone numbers.

10.4 CATI Specification Program for Study Management

A survey specification consists of a number of parameters describing when and how a survey should be executed. The parameters for the survey definition are held in a CATI specification file that is created using the CATI Specification Program (`btspec.exe`). The survey specification includes, for example, the period for the survey, a specification of the days on which interviews will be held, and the (number of crews of) interviewers. The survey specification also indicates which treatment must be given to telephone numbers in several situations. For example, how many times a number must be called back if it is busy or if there is no answer. You can also select fields to appear on the dial menu and define time zones and slices. The specification file is saved with a `.bts` extension. You must have a specification file before you can run the data model and the CATI Management Program.

The survey specification file can be protected by a password. Authorised persons (those who know the password) can change the survey specification, if need be. It might happen, for instance, that more interviewers are needed on a day or that interviewing has to start earlier than originally planned. People who do not know the password are allowed to look at the survey specification but cannot change it.

You can use the CATI specification file in other locations or for other surveys. This provides yet another way to define specifications for different situations using the same data model.

Changing the data model does not affect the CATI specification file, unless one of the fields used in the definition is deleted or renamed. In that case, the survey definition ignores entries relating to the old field name.

The CATI specification file must be complete before the CATI Call Management system can be used, but you can change the CATI specification file while interviewing is occurring. Each time a new number is presented to an interviewer, the Data Entry Program (DEP) checks whether the specification file has changed, and if it has, the DEP will reread the specifications from it.

This section describes how to create the specification file and how to set parameters for it. Once you create this file, you can edit it during the survey.

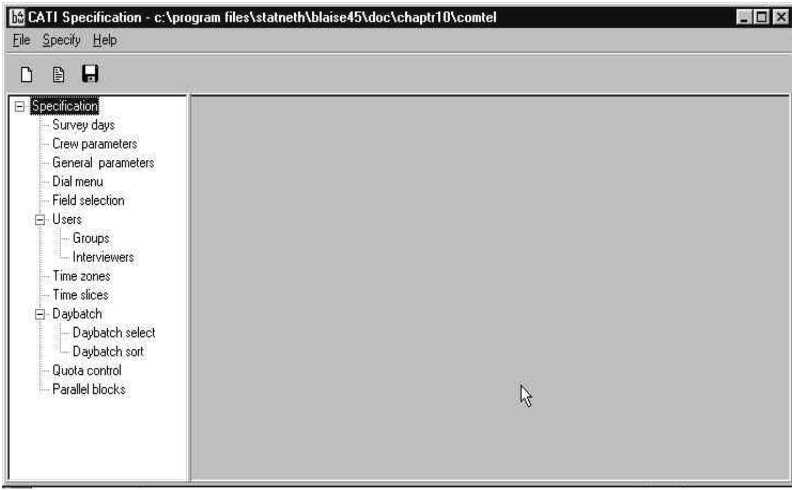
10.4.1 Create a specification file

To create a specification file, you must first have a data model that has been prepared. The CATI Specification Program reads information from the data model's meta information file (which has a `.bmi` extension).

Open the Control Centre. Select *Tools ➢ CATI Specification* from the menu and the CATI Specification window appears without a file. Then select *File ➢ New* from the menu and open your data model's meta information file, which has a `.bmi` extension.

You can also open your data model in the Control Centre and then select *Tools ➢ CATI Specification*. The program will try to open the specification file with a name that can be determined based on the active window. If that specification is not found, you will be prompted to create a specification for the open file. The system checks whether the data model definition is suited for CATI. The `btspec.exe` program can also be run separately outside the Control Centre.

Figure 10-8: CATI Specification Program



The window contains an explorer view to define all aspects of the survey. Select an item in the left pane and choose the appropriate settings for your survey as described in the following sections. The views that are available depend on whether there is a `.bmi` file and what settings have been set.

Minimum requirements

At minimum, you have to have the following for the specification file to be considered complete:

- Survey days have to be set.
- The *Dial menu* if it is not to be skipped must contain at least one entry that has the treatment *questionnaire.*
- *Field Selection* must have at least one field with the function *telephone*.

Save the specification file

To save your settings, save your file with a `.bts` extension. You will need this file to conduct CATI management activities and to run the survey.

Set a password

You can protect all your settings by creating a password. Select *File ➤ Password* from the menu. You can protect your survey specification from inadvertent changes by giving it a password. Only supervisors who know the password will be allowed to change the specification. The password is also asked if you want to create a day batch. If a password has been entered previously, then the program asks for it before it can be changed. If the old password has been entered correctly, a new password can be entered. Then you are asked to retype the new password to verify it. If the verification fails the new password will not be accepted.

A password may consist of up to ten characters. If you want to clear a previously entered password, you must first enter the old password correctly and then enter an empty string.

If a survey specification has a password, then if something has been changed the program will ask for it before WRITING the data to a survey specification file. It does NOT ask for the password before READING a survey specification file. In this way it is always possible to look at a survey specification, but it cannot be changed by accident or by unauthorised people.

The program will only ask once for the password. If you have entered the password correctly once, it will not ask for it again during the same run.

Print the specification file

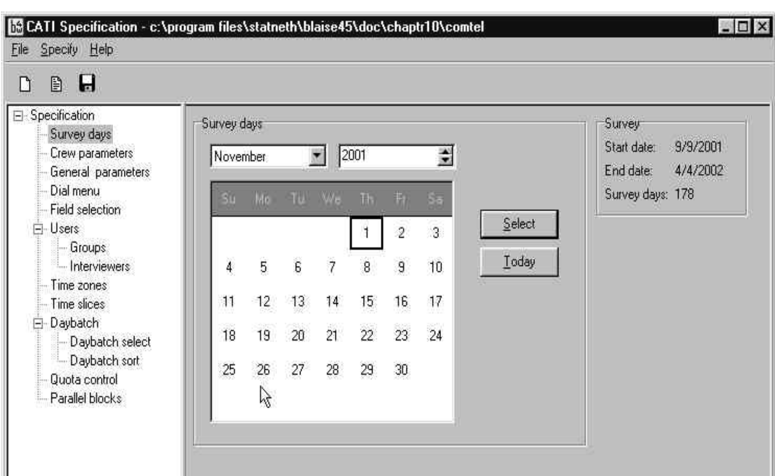
You can print the parameters from your file. Select *File ➤ Print* from the menu and the *Select topics to print* dialog box appears. Select the items you want to print and then click the *Print* button.

10.4.2 Survey days

Before you can do anything else, you must define the days of your survey.

Select *Survey days*.

Figure 10-9: Survey days branch

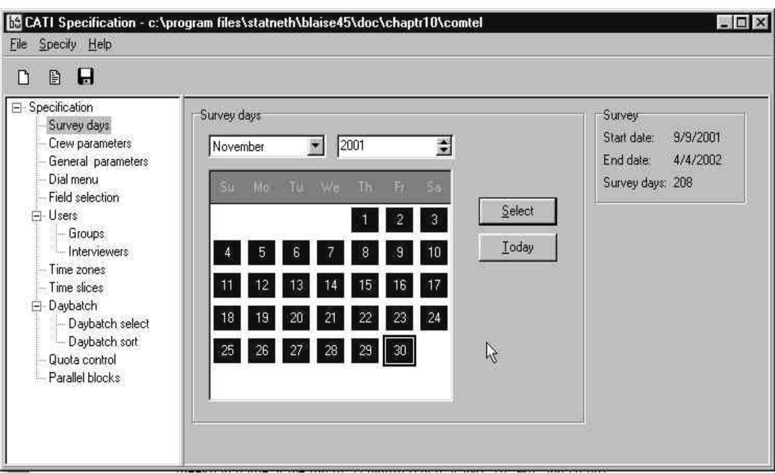


Select the month and year from the boxes above the calendar. Then select the exact days for the survey by double-clicking the numbers on the calendar, using the space bar to toggle, or placing the cursor on a day and clicking the *Select* button.

You can select and deselect all available weekdays by double-clicking the weekday name at the top of a column (such as *Mo, Tu, We,* and so on).

A day is selected when a highlighted box appears around the number. The start and end dates of the survey then appear in the *Survey* section on the right. For example, the following figure shows all weekdays selected in June 2001.

Figure 10-10: Selected survey days



Continue to set survey days by choosing the appropriate month, year, and days.

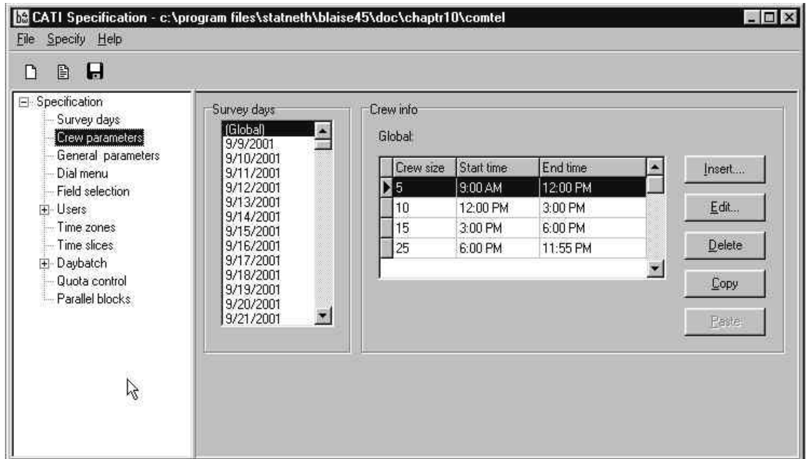
! You can clear all old survey days (days before today) by pressing Ctrl-C. This can be useful if you re-use an old specification file that had a previous survey period.

10.4.3 Crew parameters

For each active day you can specify the crew size and working hours for a maximum of five crews. Crew sizes are only used for the purpose of distributing all-day (no time or time range) soft and medium appointments throughout the calling day.

To set crew parameters, select *Crew parameters*.

Figure 10-11: Crew parameters branch

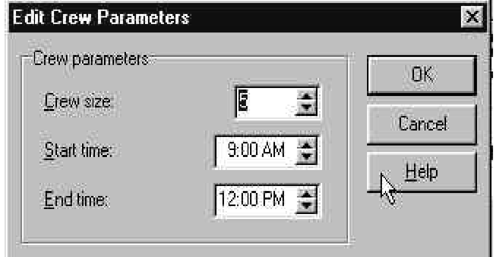


The active days of your survey appear on the left. For each crew you can specify the number of interviewers and the start and end times. If you do not specify any day parameters, the CATI Management Program will assume that there is one crew working from 9:00 am until 9:00 PM.

First, select the survey days from the list on the left. To set the crews for all days, choose the *(Global)* option. Global crew definitions become the default for the days for which you have not defined any specific values.

Then either edit the default settings by clicking the *Edit* button or add a new crew by clicking the *Insert* button. The *Insert* or *Edit Crew Parameters* dialog box appears.

Figure 10-12: Edit Crew Parameters dialog box



Indicate the crew size, start time, and end time. You do not have to specify the number of interviewers, but you must include start and end times for the crew to be valid. You can have more interviewers than the number entered.

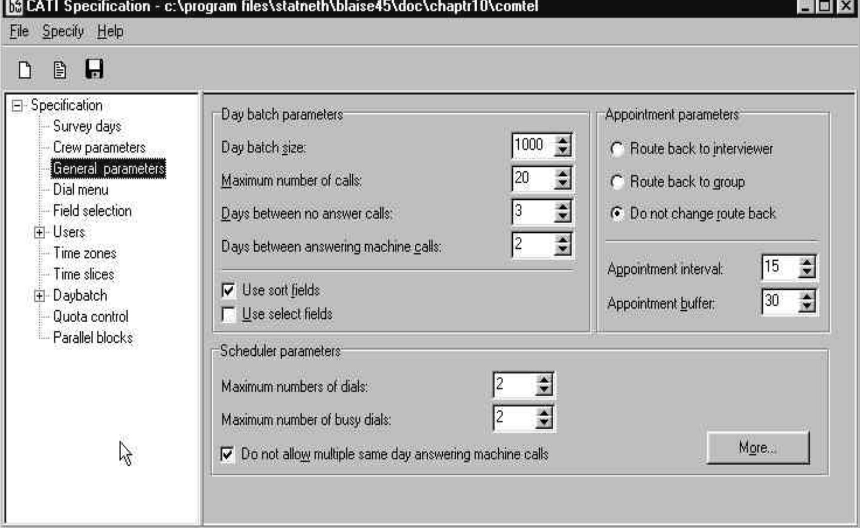
Click the *OK* button to return to the *Crew parameters* dialog. You can also copy, paste, and delete definitions by clicking the appropriate buttons.

When appointments are made during interviewing, the system will only accept appointments that fall within the global start and end times or the times specified for a particular day. For period appointments and day-of-the-week appointments, only the global times are taken into account. For this reason you should define all of the valid dates and times of the survey and crews right at the start of the survey. This way you avoid appointments for times when you will not have a crew at work.

10.4.4 General parameters

Set parameters for the daybatch, appointments, and scheduler by selecting the *General parameters* branch.

Figure 10-13: General parameters branch



Set daybatch parameters

A daybatch is a file that contains a set of forms for respondents who may be contacted on a specific day in the survey period. Daybatch parameters influence

the construction of the daybatch and thus how you can manage the survey. They take effect when the daybatch is created.

Set the following parameters for the daybatch:

- *Daybatch size*: Specify the maximum number of forms to be included in the daybatch. It is incremented by hundreds.
- *Maximum number of calls*: Specify the maximum number of calls that can be made for a telephone number. If the maximum has been reached, the number will no longer be included in the daybatch. This setting is ignored for numbers with a hard appointment or with a preference appointment that is current. Note that a call may consist of more than one dial.
- *Days between no-answer calls*: Specify how many days to wait before including the number again in a daybatch if the last dial result was *No-answer*. For numbers with a hard or preference appointment that is current, this setting is ignored.
- *Days between answering machine calls*: Specify how many days to wait before including the number again in a day batch if the last dial result was *answering service*. For numbers with a hard or preference appointment that is current, this setting is ignored.
- *Use sort fields:* Select to sort the forms in the daybatch based on the fields selected on the *Daybatch sort* tab. This controls the order in which forms are placed in the daybatch.
- *Use select fields:* Select to include forms in the daybatch based on the values stored in fields selected on the *Daybatch select* tab. This controls which forms are placed in the daybatch.

! Using *Select fields* with the *Include* option has the effect of excluding forms that do not satisfy the inclusion criteria. For example if you include EST time zone forms, all other time zones would be excluded.

Set appointment parameters

The appointment parameters control the way appointments can be made with a respondent. Set the following appointment parameters:

- *Route back to interviewer:* Select to have the contents of the route-back field updated with the name of the interviewer who made the appointment, thus routing the form back to that interviewer.

- *Route back to group*: Select to have the contents of the route-back field updated with the name of the main group the interviewer belongs to, thus routing the form to that group.
- *Do not change route back:* Select to leave the contents of the route-back field unchanged.
- *Appointment interval*: Specify the minute intervals for setting appointments. For the number of minutes, you can choose any multiple of 5 from 5 to 60. For example, if the appointment interval is 15 minutes, then appointments can be set in steps of a quarter of an hour.
- *Appointment buffer*: Specify until how long before the end of the survey day you want the system to accept appointments. For example, if this is set to 30 minutes and the last crew stops working at 9:00 PM, the system will accept appointments up to 8:30 PM. This prevents appointments from being made just before the end of the day when it might be difficult to complete the interviews before the end of the day.

Set scheduler parameters

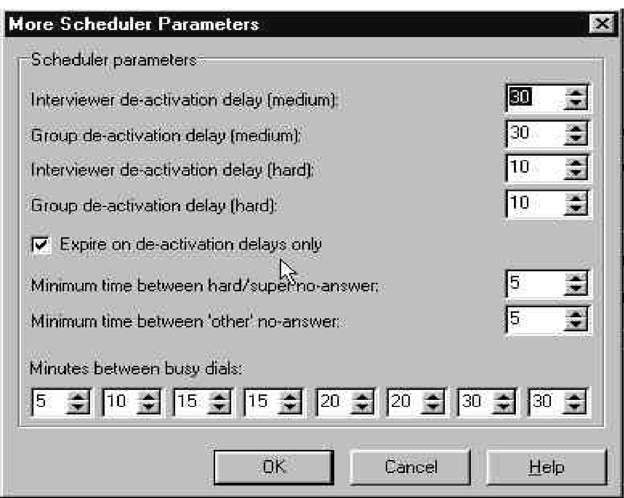
The scheduler parameters control the behaviour of the scheduler. These ensure that numbers are dealt with according to their priorities and that appointments are met on time.

Set the following scheduler parameters:

- *Maximum number of dials*: Specify the maximum number of dials that are allowed within the same call. When there has been no contact after this number of dials, the number will not be delivered again on the same day.
- *Maximum number of busy dials:* Specify how many times to redial a number if it is busy on the first dial and continues to be busy on subsequent attempts. When the maximum is reached, the number of dials is increased by one and the scheduler no longer attempts to chase this series of busies.
- *Do not allow multiple same day answering machine calls*: Check to set the status of the form to *no need today* when the dial result is set to answering machine. When not checked the number will receive the standard no answer treatment when the dial result is set to answering machine.

For more scheduler parameters, click the *More* button and the *More Scheduler Parameters* dialog box appears.

Figure 10-14: More Scheduler Parameters dialog box



- *Interviewer de-activation delay (medium)*: Specify how long the scheduler has to reserve a form for a specific interviewer in case of a medium priority appointment. If this time expires, the form will be given to the interviewer's main group. If the interviewer does not belong to a group, the form will be given to anyone.
- *Group de-activation delay (medium):* Specify how long the scheduler has to reserve a form for a specific group of interviewers in case of a medium priority appointment. If this time expires, the form will be given to any interviewer.
- *Interviewer de-activation delay (hard)*: Specify how long the scheduler has to reserve a form for a specific interviewer in case of a hard appointment. If this time expires, the form will be given to the interviewer's main group. If the interviewer does not belong to a group, the form will be given to anyone.
- *Group de-activation delay (hard):* Specify how long the scheduler has to reserve a form for a specific group of interviewers in case of a hard appointment. If this time expires, the form will be given to any interviewer.
- *Expire on de-activation delays only:* Do not check if you want the default situation, in which de-activation delays are applied only if the specified interviewer (in the case of interviewer de-activation delays) or a member from the specified group (in the case of group de-activation delays) is currently running the DEP. If you want the de-activation delays to be honoured irrespective of who is on the system, be sure to enable (check) this option. For example, if there is one interviewer assigned to the *Spanish* group and that person is delayed in getting to work on a particular morning,

appointments assigned to that interviewer or to the *Spanish* group would immediately (i.e., without the specified delays) be given to other interviewers because the target interviewer is not logged in if the option is not enabled. In this example, with the option unchecked, you would have English-speaking interviewers suddenly presented with Spanish forms whereas you probably would want the deactivation delays to govern.

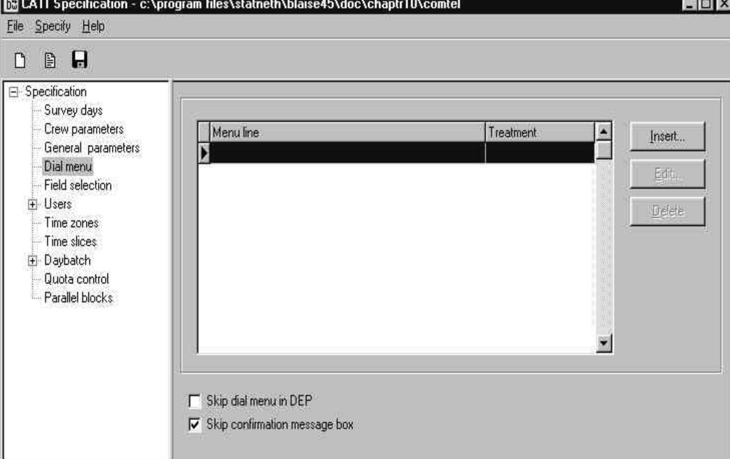
- *Minimum time between hard/super no-answer:* Specify how long the scheduler has to wait between two consecutive dials of a number that has a hard/super appointment.
- *Minimum time between 'other' no-answer:* Specify how long the scheduler has to wait between two consecutive dials for a number that does not have a hard/super appointment.
- *Minutes between busy dials:* Specify how long the scheduler will wait between consecutive busy dials. If the maximum number of busy dials is five, the system will use the first four intervals.

10.4.5 Dial menu

When you define a CATI survey, you may want to create a dial menu. The dial menu then appears in the DEP window whenever interviewers request a new form. For each menu item, you can specify the menu text and the dial result to be attached to that text.

To create the dial menu, click the *Dial menu* branch.

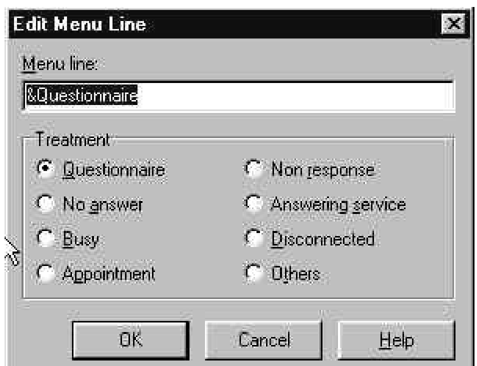
Figure 10-15: Dial menu branch



The options set here determine what the dial menu looks like during interviewing.

To insert a new option in the menu, click the *Insert* button. To edit an entry, click the *Edit* button. The *Insert* or *Edit Menu Line* dialog box appears.

Figure 10-16: Edit Menu Line dialog box

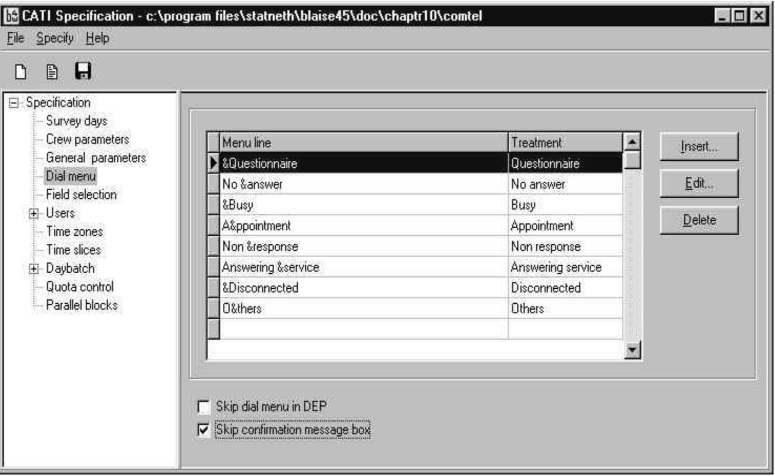


In the *Edit Menu Line dialog* box, type the text that will appear in the dial menu. To use a letter as a short-cut key, type an ampersand (&) in front of that letter. If a shortcut letter occurs more than once, the first option where it occurs will be activated.

You must select one of the eight possible dial results for each menu line. The dial result selected on the dial menu will be the dial result associated with the dial attempt unless overridden by code in the Blaise instrument. If you wish to go from a menu line directly into the main instrument, you must associate the questionnaire treatment with the menu line. If you have associated a parallel block with a treatment (see section 10.4.11), and a menu line is associated with the same treatment, you will automatically go to the parallel block when selecting the menu line.

When finished, click the *OK* button. The following sample shows a completed *Dial menu*:

Figure 10-17: Completed dial menu tab



All defined menu items will also appear on the dial menu for the supervisor. If you have not specified the treatments Appointment and Nonresponse, the program will automatically insert them in the dial menu for supervisors. The system will also include the option Call as soon as possible in the dial menu for supervisors. This allows the supervisor to direct a form to the first available interviewer.

If you do not wish to use the dial menu, you can disable this screen by enabling (checking) Skip dial menu in DEP option. When enabled the dial screen will not be displayed but a message box will be displayed in which the interviewer has to confirm that the next form should be delivered.

You can disable the dial confirmation message by enabling (checking) the *Skip confirmation message box.* When enabled there will be no confirmation message box when the OK button in the dial menu dialog in the DEP is clicked.

10.4.6 Field selection

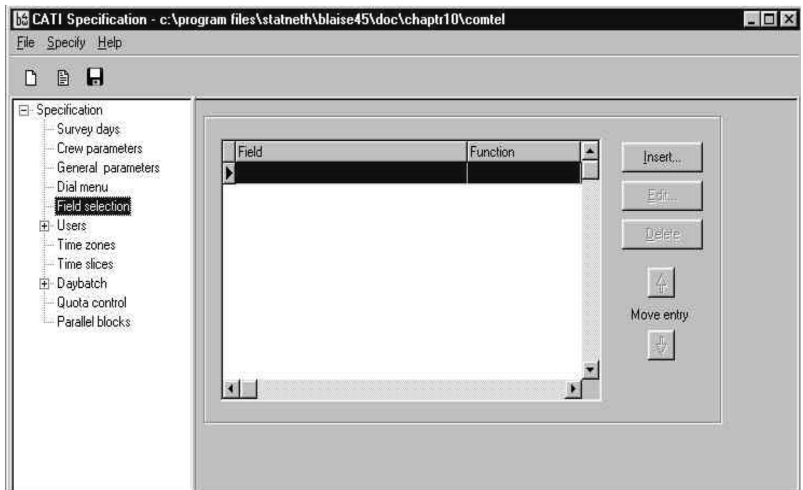
Field selection serves two purposes in your CATI survey. First, you can select the fields that you want displayed on the dial screen for interviewers, shown in the Call Management overview and/or added to the history file. For this you might select fields that provide information about the form, such as telephone number, address, or time zone.

Second, when you select fields you also tell the system which fields in your data model to use for the following functions:

- Telephone numbers (telephone field)
- Routeback information (ToWhom field)
- Time zones
- Quota criteria
- Time slices

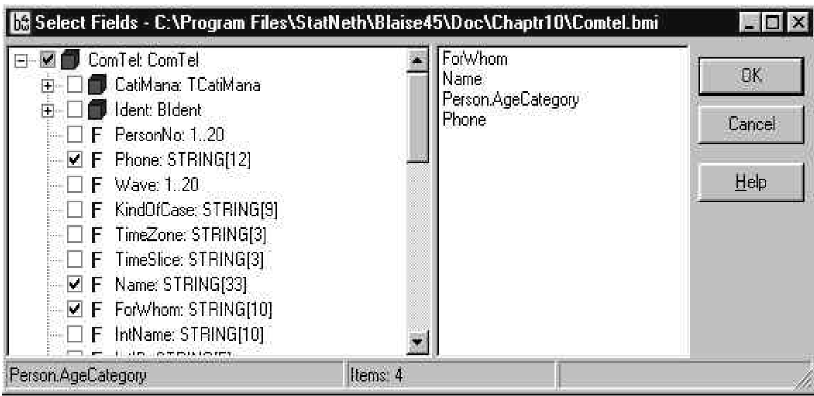
To select fields, select *Field Selection*.

Figure 10-18: Field selection branch



Click the *Insert* button. A dialog box with a tree diagram of the data model appears. This is the structure of your data model, similar to that seen in the Structure Browser.

Figure 10-19: Data model structure when selecting fields



Select fields from the tree by double-clicking in the boxes next to them (or use the space bar to toggle). Then click the *OK* button to return to the *Field selection* tab. The fields then appear in the Field column.

Next click on a field and click the *Edit* button. The *Edit Field Function* dialog box appears.

Figure 10-20: Edit Field Function dialog box

Here you decide where the field will display, specify if the field can be edited, and assign the appropriate function to it.

Select the functions and options as described in the following bullets:

- *None*: The field has no special function. This is the default.
- *Telephone:* The field represents a telephone number. At least one field in the selection must have this function.

- *To whom:* The field contains routeback information. This must be a string field.
- *Time zone:* The field contains the time zone identification needed for time zone correction. This must be a three-character string field.
- *Quota:* The field is used as a basis for quota for parts of the data model. You can route to or route past certain parts of the data model based on the number of responses from all of the interviewers for the quota field. Only enumerated fields can be used as quota fields. You can have more than one quota field.
- *Time slice:* The field contains the time slice set identifier. This must be a three-character string field.
- *Show in overview:* Check to show the field in the overview when browsing forms in the CATI Management Program.
- *Show in dial screen:* Check to show the field on the dial screen.
- *Edit allowed:* Check to allow interviewers and supervisors to edit the contents of the field on the dial screen. Since the new value will be stored in the Blaise data file, this option should be used with care. The telephone number is a typical field for which you might use this option. String, enumerated, integer and real fields that are not key fields can be edited.
- *Add to history file*. Check to add the value of the current field to the records written to the history file. The fields will be written in the order as they appear in the fields selection list. Be aware that changing the order in the field selection list influences the order of the fields in the history file.

Click the *OK* button to return to the *Field selection* dialog.

Click the arrows to move the field up and down in the list to affect the order in which the fields appear on the dial screen.

The telephone number field will always appear first on the dial screen, regardless of where you place it on the *Field Selection* dialog.

You can scroll to the right and see the column headings Dial, List, Edit, and History with a Yes/No under each heading. A Yes under Dial means that the field will appear on the dial screen. A Yes under List means the field will appear in the CATI Management program data listing dialogs at the forms branch. A Yes under Edit means that the field can be edited and Yes under History means that the field will appear in the ASCII history file.

10.4.7 Interviewers and Groups

You can specify the interviewers who will be working for a survey. You can also create groups to help categorise the interviewers by certain criteria, such as different languages or levels of interviewing difficulty. Groups can also be used for routing back forms or to divide the daybatch into a number of sub-batches. You can create any number of groups and interviewers and assign interviewers to more than one group. If an interviewer belongs to one or more groups, you must specify a main group for the interviewer.

The names specified for the interviewers have to be identical (ignoring case) to the names that can be determined by the system for an interviewer. The system determines the name by referencing Windows® registry setting HKEY_CURRENT_USER\Environment\BlaiseUser or, if no environment setting is available, the login name.

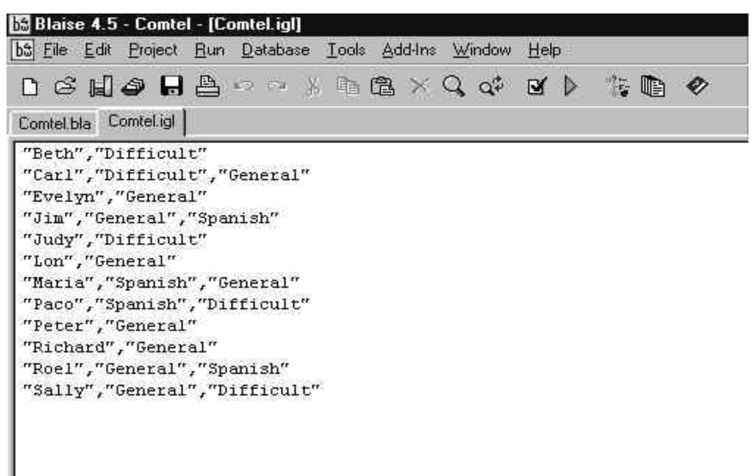
You can add interviewers and groups individually, or you can create an ASCII file and load it into the CATI Specification Program.

Add interviewers by loading a file

To add interviewers and groups by loading a file, first create a text file that lists each interviewer’s name. You can use any text editor. Interviewer names must be unique. Enclose each name in quotation marks.

To specify groups for the interviewer, follow the interviewer's name with each group to which the interviewer is to be assigned, enclosing each group name in quotation marks and separating the name and group names with commas. The following figure shows a sample of such a file in the Blaise text editor:

Figure 10-21: Sample file to load interviewers and groups

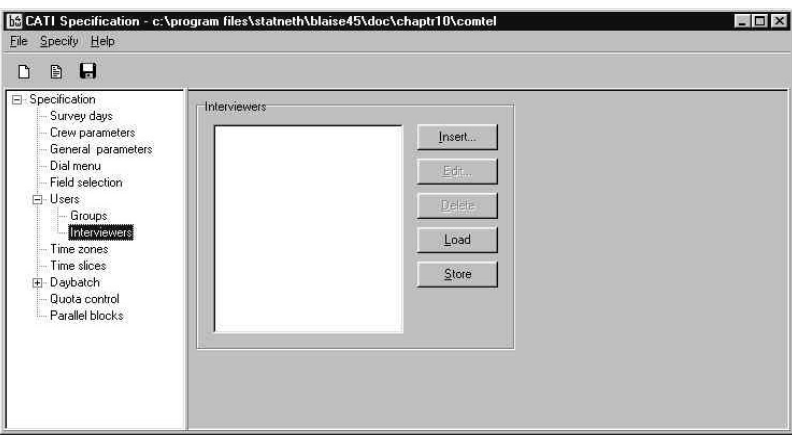


In this sample, the interviewers are assigned to the groups *Difficult, General,* or *Spanish.* The first group name to appear directly after the name will be that interviewer's main group. For example, Paco is assigned to two groups, but the Spanish group is his main group. Group names and interviewer names must all be different.

Save the file as an ASCII text file in the same folder as the data model and specification file. The file name must match the survey name exactly, but have the extension `.igl`.

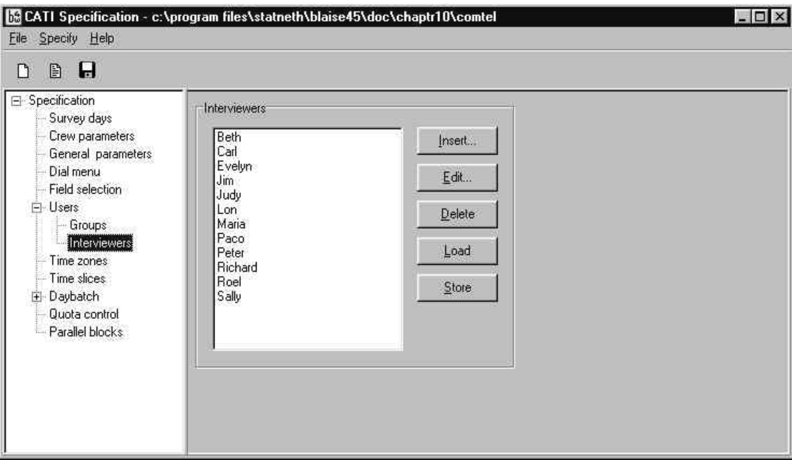
Then open the CATI Specification Program and select the *Users/Interviewers* branch.

Figure 10-22: Interviewers settings



Click the *Load* button and the interviewer list appears in the box, as shown in the following figure:

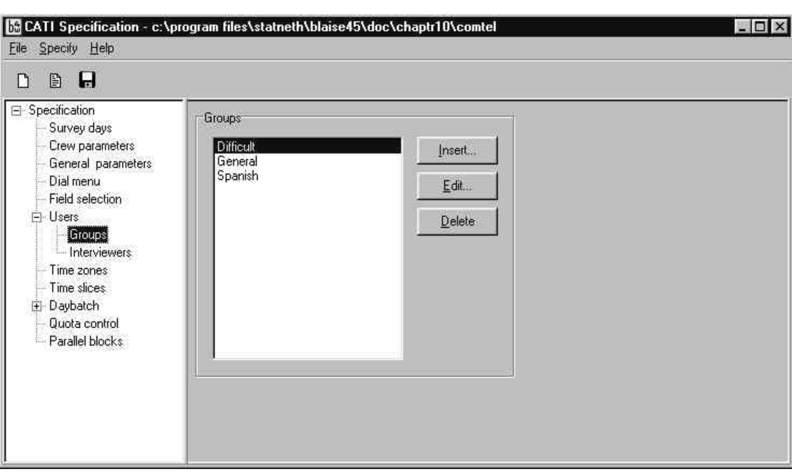
Figure 10-23: Interviewer list loaded



Click the *Store* button to save the list in the specification file.

The groups should also be assigned to the interviewers. Check this by selecting the *Groups* tab. You will see that the group names are listed. The following figure shows the *Groups* tab for our example:

Figure 10-24: Groups settings



Edit interviewers or groups

To edit a name or its group assignment, select the name and then click the *Edit* button. The *Edit* dialog box appears for either the interviewer or the group.

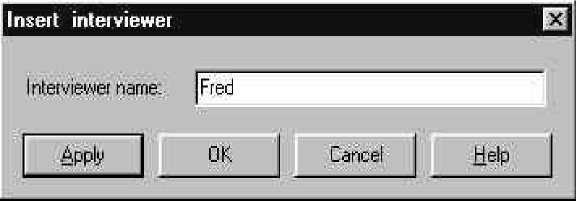
Figure 10-25: Edit interviewer dialog box

Here you can edit the interviewer or group name, change the group assignments, or change the main group. Click the *OK* button to return to the *Interviewers* settings.

Add interviewers or groups individually

To add interviewers or groups individually, first select either the *Users/Interviewers* or *Users/Groups* branch. Then click the *Insert* button and the *Insert interviewer* or *Insert group* dialog box appears.

Figure 10-26: Insert interviewer dialog box



Type the name of the interviewer or group. If you are adding several entries, click the *Apply* button and continue adding. When you have typed your final name, click the *OK* button. You can edit the entries as described above.

Example: Groups mutually exclusive

You might have some interviewers you do not want handling default interviews if there are enough difficult ones to keep them busy. In this case, declare only the groups *Difficult* for those interviewers. They will not get any default interviews unless a hard or medium appointment has been made for a default form and the relevant de-activation delays have expired (or are not applicable because there are no other interviewers on the system and *Expire on de-activation delays only* has not been enabled).

Example: Groups not mutually exclusive

You can have two interviewers assigned to both the *Difficult* and *Easy* groups with the first interviewer having the *Difficult* group as his main group and the second interviewer having the *Easy* group as his main group. When the first interviewer requests a case if both difficult and easy cases with the same priority are available, a difficult case will be delivered to him because the scheduler finds such cases more suitable for him. The scheduler would find easy cases more suitable for the second interviewer in similar circumstances.

Example: Deliver forms to a specific interviewer

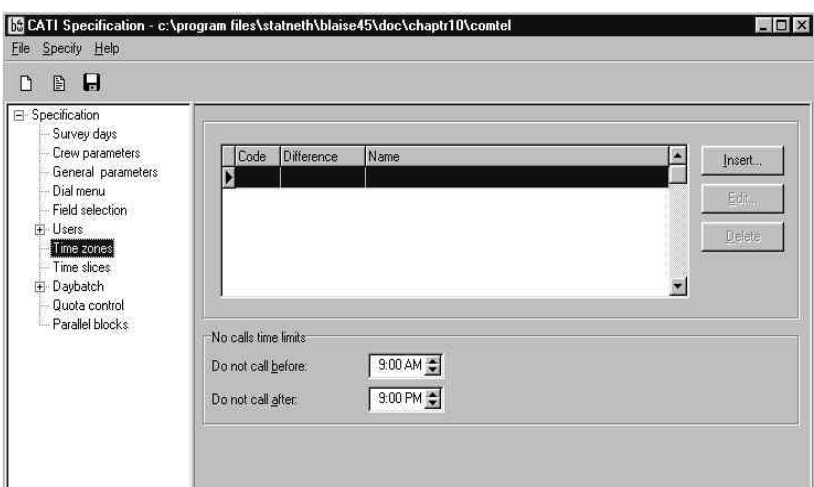
If you are using groups but want a specific interviewer to handle a set of forms, then you can put her in her own group.

10.4.8 Time zones

Time zones define the time differences between the interviewer's location and the respondent's location.

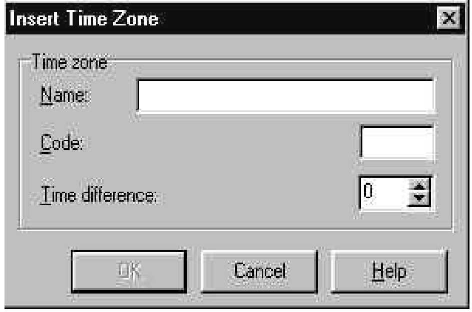
To define time zones, select the *Time zones* branch.

Figure 10-27: Time zones settings



To create a new time zone, click the *Insert* button. The *Insert Time Zone* dialog box appears.

Figure 10-28: Insert time zone dialog box



In the *Name* and *Code* boxes, specify a name and a three-letter code. The code used here must be put in the *Time zone* field of the data model.

In the *Time difference* box, specify a time difference, or offset from the interviewer's time, for each time zone. Time zone differences can be specified in

15-minute increments. Time zones in the east are given negative offsets while time zones to the west are given positive offsets. For example in the United States, a call centre operating in the Central time zone (CST) would have a negative offset of –60 for forms in the Eastern time zone (EST).

All offsets must be adjusted if the time of the interviewer's location changes. For example if a 2nd call centre operates from a different time zone the 2nd call centre requires its own offsets.

Click the *OK* button when you are finished to return to the *Time zone* settings.

Set time limits by adjusting the *Do not call before* and the *Do not call after* times on the tab. The time specified is the same for all time zones and for the local time zone. During interviewing, the call scheduler checks whether the respondent's time lies between these specified times. Except in the case of hard or super appointment, the scheduler does not deliver a form if the respondent's time does not lie between the two specified times.

10.4.9 Time slices

Time slices allow you to spread the call backs for no-answer and answering machine dials when there is no pending appointment over different parts of the survey day or week. For example, if you receive a default-priority no-answer during a weekday, you might want the next try to be an evening or weekend.

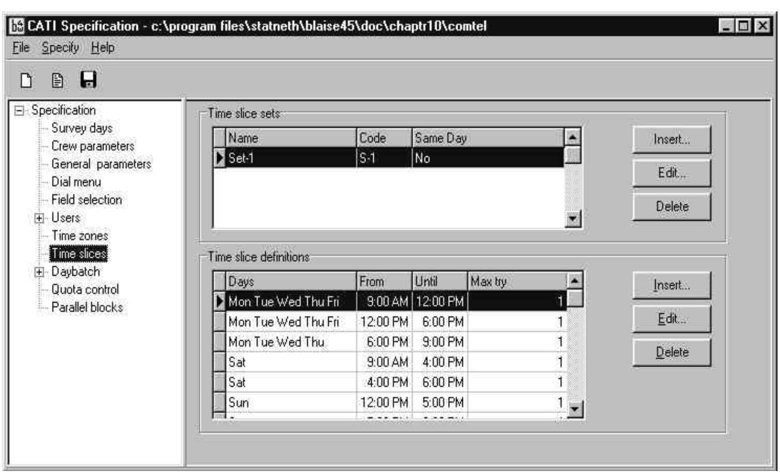
When you define time slices, the system keeps track of what time a number was called, and will dial the number in a different time slice the next time it becomes active. Time slice definitions are used only for forms that have *default* status.

You can define one or more slice *sets*. A time slice set is a group of slice definitions that belong together. If you use more than one set, you must first define a time slice field in your data model (see the *CATI Data Model* section in this chapter). If a slice field has been defined, the time slice mechanism will only be used if that field contains a valid slice set code. If no slice field has been defined, the first slice set will be used.

Define time slice

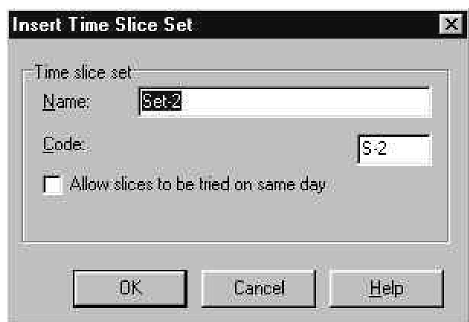
To define a time slice set, select the *Time slices* branch.

Figure 10–29: Time slices settings



Click the first *Insert* button (in the *Time slice sets* box) and the *Insert Time Slice Set* dialog box appears.

Figure 10-30: Insert Time Slice Set dialog box

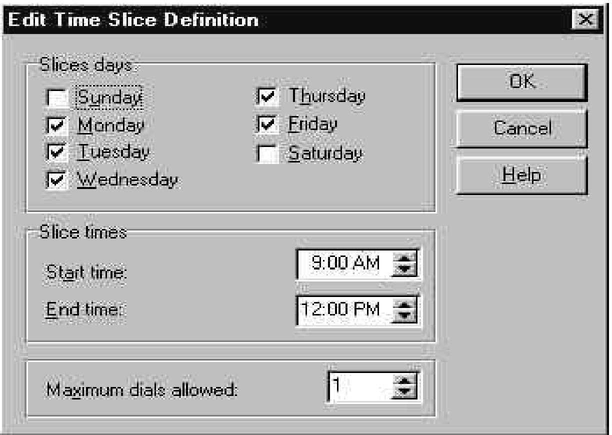


- Specify a name for the time slice.
- Specify a three-character code for the time slice set. The set codes must be unique.
- Select the *Allow slices to be tried on same day* if you want to allow the scheduler to deliver a form in different time slices during one day.

Click the *OK* button to return to the *Time slices* tab.

Next, define the time slices for the time slice set. Click the *Insert* button in the *Time slice definitions* box. The *Edit Time Slice Definition* dialog box appears.

Figure 10-31: Edit Time Slice Definition dialog box



- Select the days for the time slice. You can select more than one day.
- Select the start and end times for the time slice.
- Specify the maximum number of dials allowed in that time slice.

All time slices for a set must be mutually exclusive. Click the *OK* button to return the *Time slices* settings.

10.4.10 Quota control

You can designate fields to act as quotas and thus affect the delivery of forms. As soon as the quota for a certain stratum is reached, forms in the daybatch that belong to that stratum won’t be delivered to an interviewer. Forms that belong to a stratum that is already reached will not be included in future daybatches.

There are two ways quotas can be handled. If you know which stratum a form belongs to, the delivery of forms for that stratum stops as soon as the quota for that stratum is reached. It is also possible that the stratum the form belongs to will be determined during the interview. In this case, you can check using the Boolean function QUOTAREACHED, and then act accordingly to affect the delivery of forms.

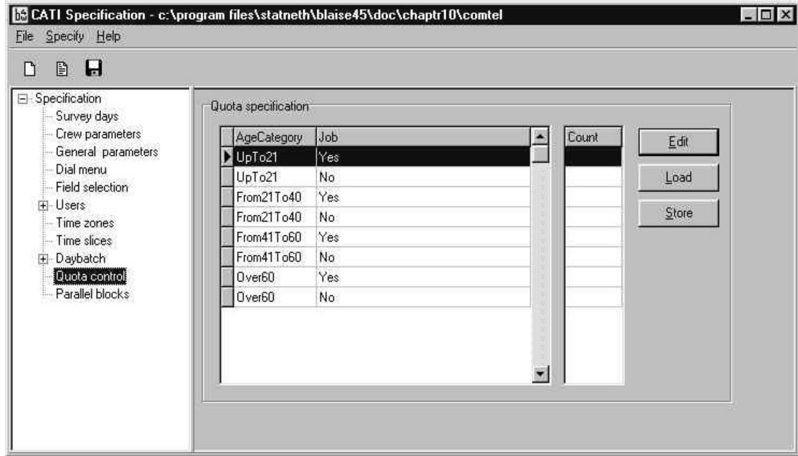
Only enumerated fields can be used as quota fields. You can set quota values for each label of the quota fields. Quota control is a two-part process:

- Designate fields for quota in the *Field selection* settings (see the *Field Selection* section above).
- Specify the quota values in the *Quota control* settings as described in this section.

You can either set quotas manually or load an ASCII text file that you prepare first.

To set quota values, select the *Quota control* branch. The labels for each field chosen for quotas are listed.

Figure 10-32: Quota control settings



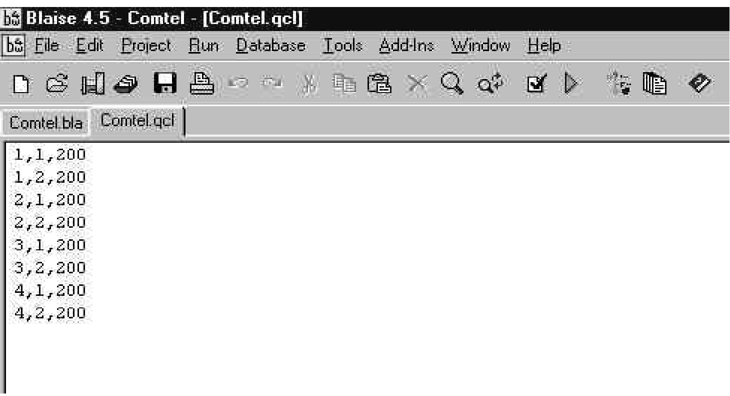
Set quota values manually

To set the counts manually, simply type the values for each quota category in the *Count* column. You can edit items by clicking the *Edit* button.

Load quotas in a file

To load quota values from a file, prepare an ASCII text file that lists the strata and the quota values. The following figure shows a sample file you might use for the quota specification shown in Figure 10-32 above.

Figure 10-33: Sample file to load quota values

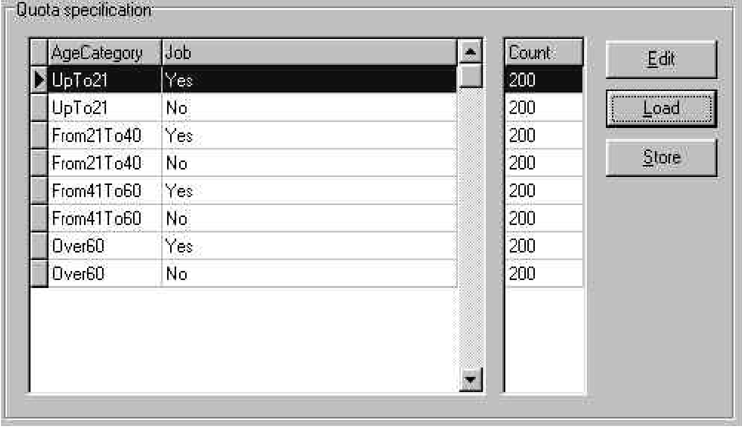


In this example, there are crossings that produce eight cells: four possible answers for *AgeCategory* and two possible answers for *Job*. In the first line of the file, (*1,1,200),* the first number (*1*) is the value of the first possible answer for the first field; the second number (*1*) is the value of the first possible answer for the second field; and the third number (*200*) is the quota value. In the next line, (*1,2,200), 1* is the value of the first possible answer for the first field, *2* is the value of the second possible answer for the second field, and *200* is the quota value.

Save the file as an ASCII file in the same folder as the data model and specification file. The file name must match the data model name exactly, but have the extension `.qcl`.

On the *Quota control* dialog, click the *Load* button and the quota counts appear in the *Count* column as shown in the following figure:

Figure 10-34: Quota specification with counts



Store the quota values

If you want to save the current quota settings in an ASCII file, click the *Store* button. The file will be saved, taking the name of the specification file and a `.qcl` extension.

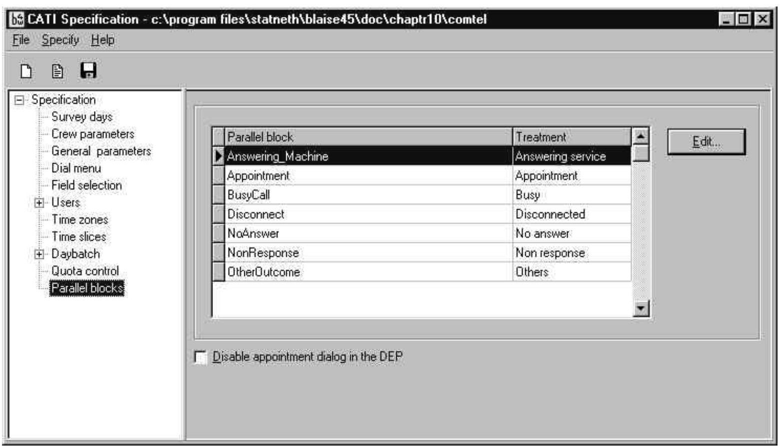
This is useful if you have a large number of strata and you need to change one of the enumerated variables that is used to define the quota. For example, you might need to add another category. If this occurs, you can't use the current quota definition and you will have to re-enter the values for all the strata. If you have a stored ASCII file, however, you can reload the previously saved quota values. All cells that were previously set will still have the stored value. All new cells will receive no value. Then you only need to enter a value for the new cells.

10.4.11 Parallel blocks

You can select parallel blocks that are in your data model and apply treatments to them through the CATI Specification Program.

Select the *Parallel blocks* branch. The parallel blocks and their treatments are listed.

Figure 10-35: Parallel blocks settings



Click the *Insert* button and the *Edit parallel block treatment* dialog box appears as shown in the following figure. Select the appropriate treatment.

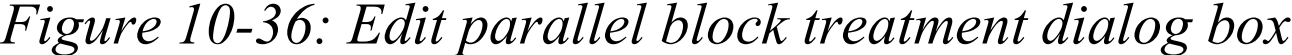
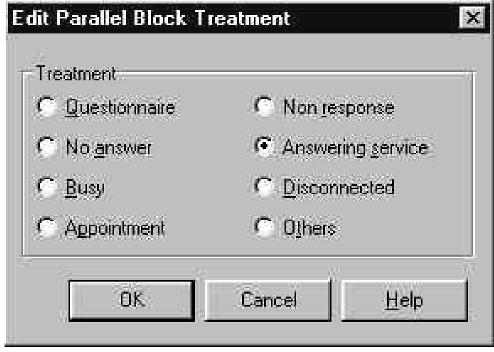


Figure 10-36: Edit parallel block treatment dialog box



If you set the treatment *Questionnaire* for a parallel block, you indicate that that parallel should be ignored by the CATI management system. The interview does not end as soon as the end of such a parallel block is reached.

Click the *OK* button to return to the *Parallel blocks* settings.

If you want to disable the Blaise appointment dialog box in the DEP, select the *Disable appointment dialog in the DEP* box. You might do this if you want to use your own appointment mechanism, either called by a DLL or programmed into your data model, instead of the default appointment dialog. You might do this to

constrain appointment dates individually for each respondent or to give a scripted appointment wording to interviewers, which cannot be done in the default appointment dialog.

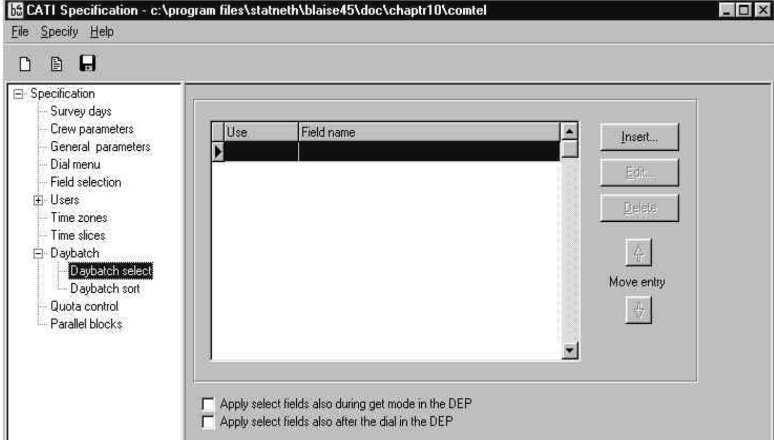
If you disable the default appointment dialog, you must still have an appointment block in your data model to record the dial result as *appointment.* But you must also impute the appropriate fields in the CATI management block that would normally be filled by the default appointment dialog. The call scheduler will then use the data to schedule and deliver forms.

10.4.12 Daybatch select

You can influence which forms are selected for the daybatch. You might do this if you want to override the daybatch selection that Blaise does automatically. For example, this may be appropriate if there are forms that the supervisor should handle, or if parts of the survey population are better targeted now than at other times. Often these fields will be those which were initialised with data before the survey began. You could also fill these fields with a Manipula program during the survey period, or use fields for which information was collected during the survey.

Select the *Daybatch select* branch as shown in the following figure:

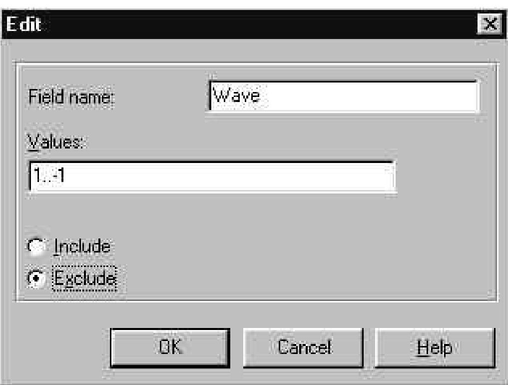
Figure 10-37: Daybatch select settings



Click the *Insert* button and the tree diagram of the data model appears. Select a field or fields and then click the *OK* button. The selected fields appear on the *Daybatch select* dialog.

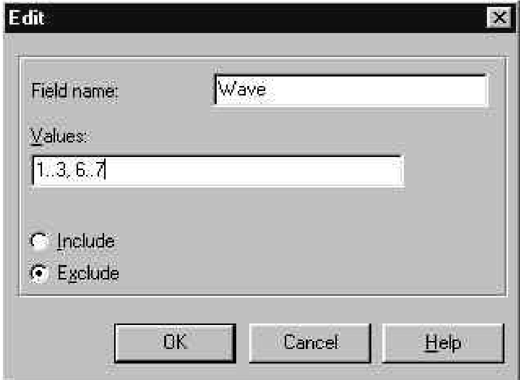
Click the *Edit* button and the *Edit* dialog box appears.

Figure 10-38: Edit dialog box for daybatch select



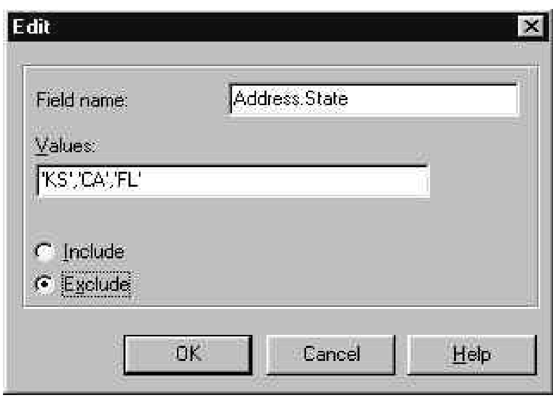
In the *Values* box, specify the values for the field on which the selection will be based. For integers, use the double dot notation to state a range, and separate entries with a comma. For example:

Figure 10-39: Value box for integers



For string fields, specify values in single quotes and separate entries with a comma. For example:

Figure 10-40: Value box for string fields



Click the *Include* or *Exclude* button to include or exclude the form based on the field value.

When finished, click the *OK* button.

Move the entries up or down the list by clicking the red arrow buttons.

! In order to have your criteria applied, make sure the *Use select fields* option is checked in the *General parameters* settings.

! Using *Select fields* with the *Include* option has the effect of excluding forms that do not satisfy the inclusion criteria. For example if you include EST time zone forms, all other time zones would be excluded.

When selecting a form in the DEP (with the get or browse) during CATI mode, the system can check the daybatch select fields to find out if the form may be added to the daybatch.

Check *Apply select fields also during get mode in the DEP* if you want to use the select fields also in the data entry program. When selecting a form through get or browse forms, the system will check the value of the daybatch select fields to determine if the form could be placed in the daybatch. If yes, access to the form is allowed in the DEP. If no, a message is displayed to the interviewer and access to the form is not allowed.

Check *Apply select fields also after the dial in the DEP* if you want to make sure that forms that are handled by the DEP and are not currently part of the daybatch, will not be added by the DEP to the daybatch when the select field indicate that

the forms should not be included in the daybatch. If *Apply select fields also during get mode in the DEP is not checked,* by using get or browse forms you are able to run the DEP on forms that, based on the values in their daybatch select fields, could not be placed in the daybatch. However, once the DEP is run on these forms, they are routinely added to the daybatch unless *Apply select fields also after the dial in the DEP* is checked.

10.4.13 Daybatch sort

You can influence the way forms are sorted in the daybatch, thus causing certain forms that would otherwise have equal priority to be worked on first. If you want to concentrate on some forms before others, you can place them at the beginning of the daybatch where they will be called first.

You do this by selecting fields and a sorting order for the fields. You can select more than one field, but usually you only need one or a few. As with the Daybatch*Daybatch select*, you will usually select fields that already have values, either from initialisation, through a Manipula program, or from interviewing.

The procedure for sorting the daybatch is similar to that for selecting the daybatch. Select the Daybatch*Daybatch sort* branch and click the *Insert* button. Select fields from the tree diagram and click the *OK* button. Select the field in the list and click the *Edit* button. Choose *Ascending* or *Descending* sort order.

The daybatch will sort and deliver the forms according to the fields and their sort order.

In order to have your criteria applied, make sure the *Use sort fields* option is checked in the *General parameters* settings.

10.5 CATI Management Program for the Supervisor

A supervisor manages the survey using the CATI Management Program. This program is used before each survey day begins and during the survey itself. You create a daybatch, which determines which forms will be worked on that day. You can view just the forms in the daybatch, view and browse all forms, and see details on a specific form. You can view specific forms as well as apply a specific treatment to a form. You can overrule the scheduling system, change priorities,

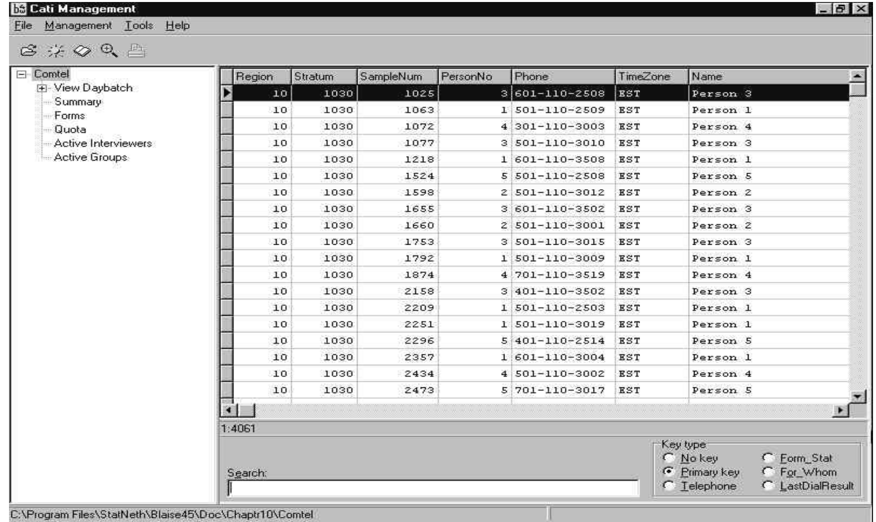
and assign forms to a specific interviewer or group. You can also review a history file and a log file.

Starting the CATI Management Program

To start the management program from the Control Centre, select *Tools* ➤ *CATI Management* from the menu. The *CATI Management* window appears.

Open a CATI specification file with a `.bts` extension, in this example *Comtel.bts*. The following sample shows the `comtel.bts` file in the CATI Management window.

Figure 10-41: CATI Management Program

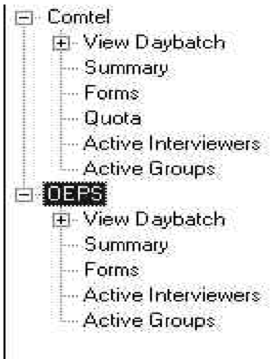


The functions are in the tree on the left and the forms of the survey are on the panel on the right. The fields you see here are those selected as *Show in overview* in the *Field selection* settings in the CATI Specification Program.

Select functions by either clicking on the tree or selecting menu options. The following sections describe the capabilities.

You can open more than one specification file at a time in order to monitor several surveys at once. When two specification files are open, you will see the name of both files on the left, with tree options for both.

Figure 10-42: CATI Management Program with two surveys



10.5.1 Create daybatch

A daybatch is a selection of forms from the entire Blaise data file that will be worked on that day. Since the CATI system cannot work without a daybatch, you have to create a daybatch for each active day in the survey period. Since you can have only one daybatch at a time, you cannot create several daybatches in advance. You would not want to do this anyway, since each time a daybatch is created, the system accounts for appointments made up to that day and other call history. A daybatch is valid only on the day for which it is created. (Note that it is always possible to create a daybatch for the next calling day, something you might want to do at the end of a survey day.)

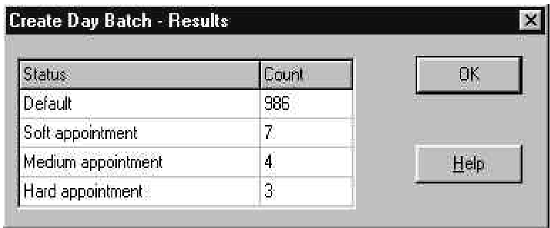
The daybatch is a binary file that only the Blaise system can read. However, when generating the daybatch file, another file is created containing the ASCII representation of the daybatch. This file has the .tdb extension and is overwritten each time a daybatch is created, or each time you access the *View Daybatch/Browse* branch of the management program. In this example the ASCII daybatch file is Comtel.tdb. It is the Comtel.tdb file that is viewed in the *View Daybatch/Browse* data listing dialog.

To create a daybatch, select *Management ➢ Create Daybatch* from the menu. The *Create Day Batch* dialog box appears with the next available survey date displayed in the box:

Figure 10-43: Create Day Batch dialog box

Click the *OK* button. The system creates the daybatch and, when finished, a dialog box appears with the results of the daybatch.

Figure 10-44: Create Day Batch Results



This box shows how many soft, medium, and hard appointments are set for the current day.

You can automate the process of creating daybatches by selecting the option *Auto create daybatch* in the Environment options of the CATI Management Program. This will create the daybatch automatically when the specification file is opened and no valid daybatch exists for the current day.

View Daybatch branches

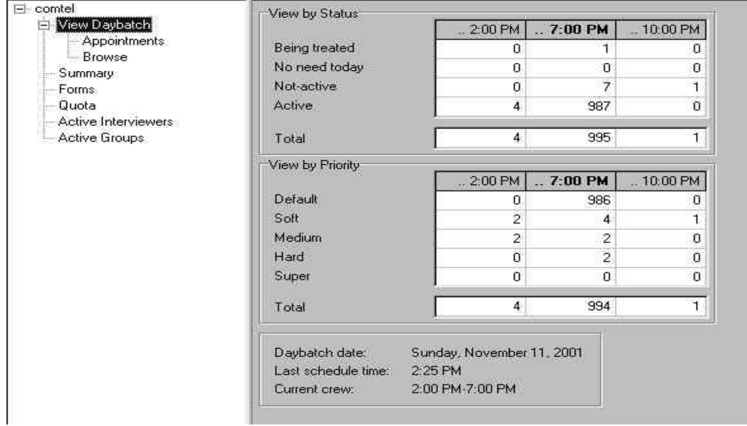
There are three branches in this part of the CATI Management tool. They are *View Daybatch*, *View Daybatch/Appointments*, and *View Daybatch/Browse*. The information in the first two branches is based directly on the binary daybatch file and is automatically updated. The information in *View Daybatch/Browse* is based on the ASCII version of the daybatch file. This latter file will be updated every time you press <Shift-F5> while having the having the focus on the viewer. It will also be updated if you physically leave the *View Daybatch/Browse* branch and re-enter it. *For* all three branches, only daybatch forms are represented. In this example there are 1000 forms in the daybatch out of 4061 in the Blaise data set.

View Daybatch

Once it is created, you can view a summary of the daybatch at any time to see the status of the interviewing. Because the information is refreshed automatically, this option provides a real-time monitor for the operation of the CATI system.

Open the CATI Management tool, then click on the *View Daybatch* branch of the tree. The daybatch appears on the right panel.

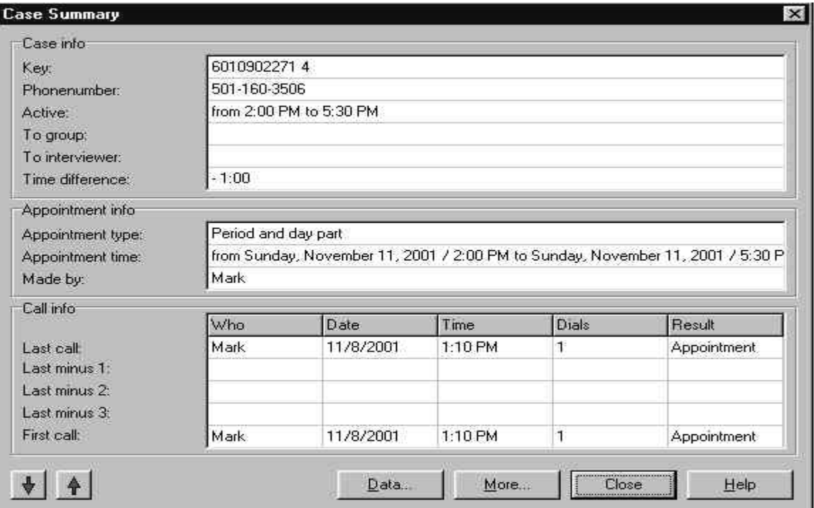
Figure 10-45: View daybatch



The panel shows two views: first by status, then by priority. There is one column for each crew shift listed in the CATI Specification file.

To view more details about a group, double-click the cell or click the cell once and click the *Zoom* speed button. The *Case summary* box appears:

Figure 10-46: Case summary box



The screen is divided into three sections: case information, appointment information, and call information.

- The case information lists the value of the primary key (if there is no primary key, an internal key is shown between brackets); the telephone number; the time interval in which the number will be active in the daybatch; the group to whom the form should be routed (if the de-activation time has expired, the text *Expired* appears); the interviewer to whom the number will be routed (if the de-activation time has expired, the text *Expired* appears); and the time difference from the interviewer's time.
- The appointment information lists the type of appointment, the time for which the appointment has been made, and who made the appointment.
- The call information lists the interviewer identification, the date and time of the last dial, the number of dials, and the result of the last dial for the very first and the last four calls.

The times listed refer to the interviewer's time, not the respondent's time.

Click the red arrows to scroll through the forms.

View more information by clicking the *More* button, or field selection information by clicking the *Data* button. The following figures show the *More info* and *Field selection data* dialog boxes:

Figure 10-47: More info dialog box

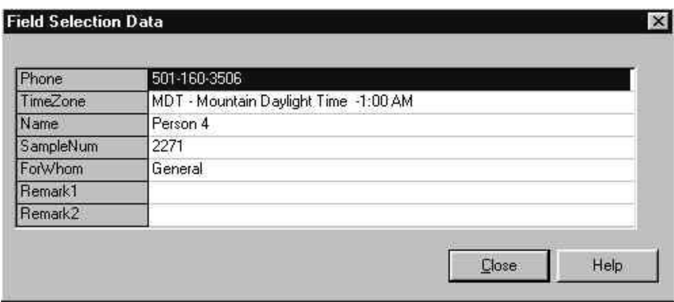
More Info

More info

Daybatch date:	Sunday, November 11, 2001
In daybatch:	Yes
First call:	Thursday, November 08, 2001
Statuscode:	Not-active
Future statuscode:	Medium appointment
Number of calls:	1
Number of dials:	0
Number of busy dials:	0
Dial time:	
Time difference:	- 1:00

Remove | Activate | Close | Help

Figure 10-48: Field selection data dialog box



Field Selection Data

Phone	501-160-3506
TimeZone	MDT - Mountain Daylight Time -1:00 AM
Name	Person 4
SampleNum	2271
ForWhom	General
Remark1	
Remark2	

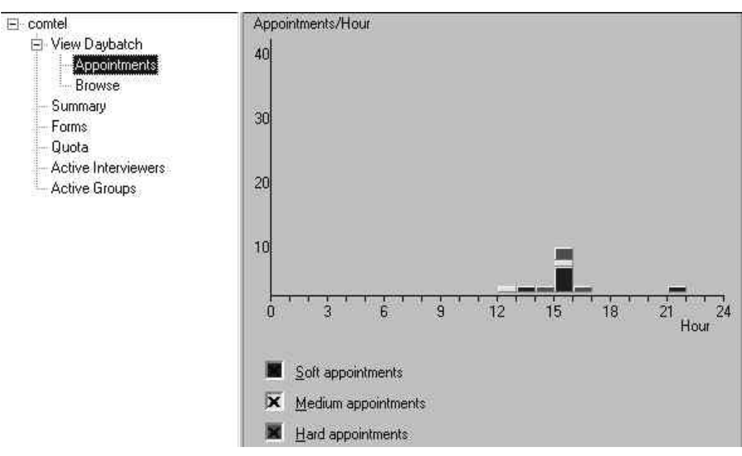
Close | Help

! The *Case summary, More info,* and *Field selection data* dialog boxes can be displayed wherever the *Zoom* function is available in the CATI Management System. They provide an excellent source of information on the survey forms.

View appointments

You can view all appointments for the current day by clicking the *Appointments* branch.

Figure 10-49: View appointments



The appointments are shown in a graph, with the hours across the bottom and the number of appointments on the left. A hard appointment for 10:45 AM is included in the 10-11 interval, while a medium appointment for 8:30 AM to 11:30 AM is included in the 8-9 interval. You can see the exact time or the starting time of all the appointment within an interval by right clicking on the interval.

Select to see soft, medium, or hard appointments by checking the boxes. To see appointment details, right -click one of the coloured bars and the Appointment details box appears:

Figure 10-50: Appointment details

Hour	Count
15:00	1
15:05	0
15:10	0
15:15	0
15:20	0
15:25	0
15:30	0
15:35	0
15:40	0
15:45	1
15:50	0
15:55	0
Total	2

In this example you can see the total number of hard appointments that are scheduled for the 1500-hour appointment interval. You can see the *Case Summary* box (documented above) for the forms associated with any bar by

selecting the bar with the mouse and using the *Management/Zoom* choice in the menu.

Schedule the daybatch

You can update the daybatch manually, causing it to schedule immediately. You might do this to see an update of the survey process immediately, instead of waiting for the five minute interval to end.

To do this, first click the *View Day batch* option on the tree. Then select *Management ➢ Schedule* from the menu. A dialog box appears for you to confirm the schedule.

Browse the daybatch

You can browse forms in the daybatch as well. Click the *Browse* branch of the tree, and the forms appear on the right. An overview of the priority of the telephone numbers in the daybatch appears:

Figure 10-51: Browse the daybatch

JoinID	Status/Priority	StartInterval	EndInterval	FuturePriority
3679	Not-active	16:00	22:00	Hard appointment
2989	Not-active	15:00	22:00	Hard appointment
1695	Not-active	14:45	22:00	Hard appointment
536	Medium appointment	12:00	20:00	Medium appointment
2271	Being treated	14:00	17:30	Medium appointment
2437	Not-active	15:00	17:00	Medium appointment
3121	New appointment for today	15:45	22:00	Hard appointment
3352	Not-active	15:00	17:00	Soft appointment
2411	Not-active	15:00	17:30	Soft appointment
16	Soft appointment	13:00	15:00	Soft appointment
23	Not-active	15:00	18:30	Soft appointment
2397	Not-active	21:00	22:00	Soft appointment
3835	Not-active	15:00	15:30	Soft appointment
1394	No need today	13:00	14:30	Soft appointment
1505	Default	12:00	21:00	Default
1013	Default	12:00	20:00	Default
1311	Default	12:00	20:00	Default
2441	Default	12:00	22:00	Default
133	Default	12:00	20:00	Default

Use the scroll bars to view all columns. The term default is used in the *Status/Priority* column to indicate that a form is active and hence available for a dial.

Activate non-actives

You can activate non-active forms, except those that have hard appointments or for which busies are being chased. First click the *View Day batch* branch of the tree, then select *Management ➢ Activate* from the menu. A dialog box appears for

you to confirm the activation. The current status of appropriate not-active forms will be changed to default and the forms will hence be available for interviewing.

Activating forms might be appropriate if you have soft appointment cases that would be active later in the day and you want them to be active now to keep your crew busy.

! Forms would have the status *Not-active* if they are in a more westerly time zone and the beginning calling time for that time zone has not yet been reached. If you activate forms, those in a more westerly time zone would also be activated meaning that they are eligible for calling before the normal starting time for that time zone (as set in the CATI specification tool, *Time zones* branch, *Do not call before* time).

! Non-active forms available for re-activation include those for which time slices apply. This means that a form that has been receiving only no-answers can be tried a second time in the same time slice if activated.

Status/Priority column

The *Status/Priority* column displays the status code for not-active cases and the priority of active cases. Status codes include *Being treated*, *Not-active*, *Busy*, *No-answer*, *No need today*, and *New appointment for today*. If the form has Status code *Active* the priorities you may see included: *Default*, *Hard appointment*, *Medium appointment*, and *Soft appointment*.

10.5.2 Summary

You can view a summary of all the forms in the survey. Click the *Summary* option on the tree and the *Summary* panel appears on the right. This is refreshed automatically.

Figure 10-52: Summary

Not yet done

		Number of Calls: 0	1	2	3	4	5	>5
No call	3991	3991	0	0	0	0	0	0
No answer	17	0	15	0	2	0	0	0
Busy	8	0	6	1	1	0	0	0
Appointment	19	0	15	3	1	0	0	0
Answering machine	3	0	3	0	0	0	0	0
Total	4038	3991	39	4	4	0	0	0

Done

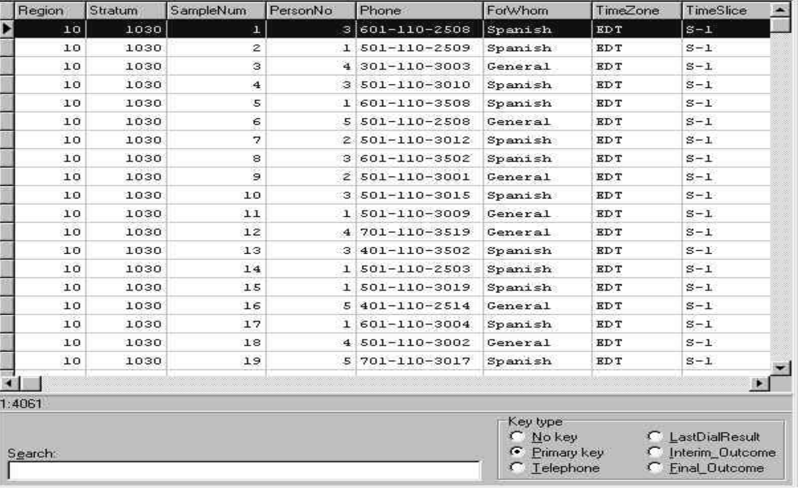
		Number of Calls: 0	1	2	3	4	5	>5
Response	12	0	12	0	0	0	0	0
NonResponse	6	0	5	1	0	0	0	0
Disconnected	3	0	3	0	0	0	0	0
Others	2	0	2	0	0	0	0	0
Total	23	0	22	1	0	0	0	0

The summary displays the outcome of all records in the Blaise data set. The summary is given by the eight high-level Blaise CATI outcome codes, as well as the row *No call*. The *Not yet done* section represents forms that are still eligible for the call scheduler (either today or in the future). The *Done* section represents forms that Blaise CATI is no longer concerned with unless action is taken (for example with Manipula) to place some forms back in the queue. The *Total* of the *Not yet done* section and the *Total* of the *Done* section add to the number of records in the Blaise data set. In both sections, the *Number of Calls* columns summarise the effort needed to reach the disposition (but note the difference between a *Call* and a *Dial* as documented above).

10.5.3 Forms

The *Forms branch* allows you to browse all the forms in the entire Blaise data file. When you click the *Forms* branch a data listing of all forms appears on the right panel:

Figure 10-53: Forms



Region	Stratum	SampleNum	PersonNo	Phone	ForWhom	TimeZone	TimeSlice
10	1030	1	3	601-110-2508	Spanish	EDT	S-1
10	1030	2	1	501-110-2509	Spanish	EDT	S-1
10	1030	3	4	301-110-3003	General	EDT	S-1
10	1030	4	3	501-110-3010	Spanish	EDT	S-1
10	1030	5	1	601-110-3508	Spanish	EDT	S-1
10	1030	6	5	501-110-2508	General	EDT	S-1
10	1030	7	2	501-110-3012	Spanish	EDT	S-1
10	1030	8	3	601-110-3502	Spanish	EDT	S-1
10	1030	9	2	501-110-3001	General	EDT	S-1
10	1030	10	3	501-110-3015	Spanish	EDT	S-1
10	1030	11	1	501-110-3009	General	EDT	S-1
10	1030	12	4	701-110-3519	General	EDT	S-1
10	1030	13	3	401-110-3502	Spanish	EDT	S-1
10	1030	14	1	501-110-2503	Spanish	EDT	S-1
10	1030	15	1	501-110-3019	Spanish	EDT	S-1
10	1030	16	5	401-110-2514	General	EDT	S-1
10	1030	17	1	601-110-3004	Spanish	EDT	S-1
10	1030	18	4	501-110-3002	General	EDT	S-1
10	1030	19	5	701-110-3017	Spanish	EDT	S-1

The fields shown are those for which the option *Show in overview* was checked on the *Field selection* branch in the CATI Specification Program.

To search for a specific form, click the key type to search on, then type the value for the key in the *Search* box. If the key consists of more than one field, separate values by semicolons.

See details on a form

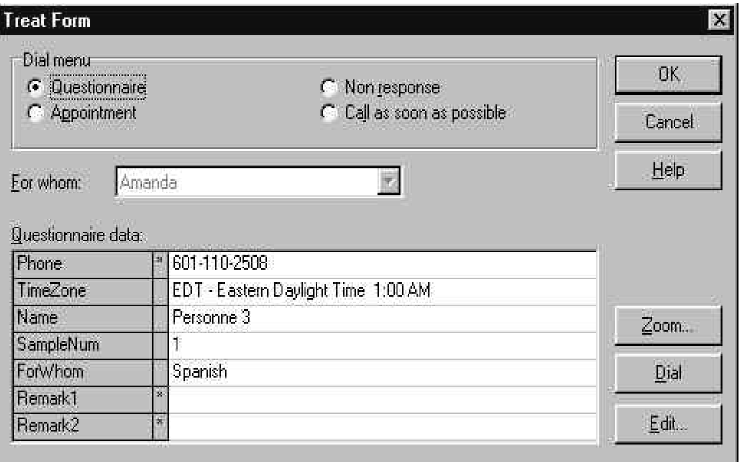
To see more information about a form, click the form and zoom in on it by selecting *Management ➢ Zoom* from the menu or clicking the *Zoom* speed button. The *Case summary box* appears. This is identical to the box that appears when you zoom on a form when browsing the daybatch. From here you can view additional information.

Select form for further treatment

There may be times when you need to apply a treatment to an individual form outside of the regularly scheduled form delivery conducted by Blaise. You would usually do this to make an appointment, complete a particular interview, or route an appointment to a particular group or interviewer.

To select a form for further treatment, select the *Forms* option on the tree, then double-click the form (or click once on the form and select *Management ➢ Select* from the menu). The *Treat Form* dialog box appears.

Figure 10-54: Treat Form dialog box



This is almost identical to the dial screen shown to interviewers.

Make changes or treat the form as described in the following bullets.

- Dial the number by clicking the *Dial* button. This option is for use with a modem.
- Under *Dial menu,* click the appropriate button to select a dial result for the number. (Note, any high-level dial result chosen in this dialog will not appear in the history file. A way to avoid this is to enter the instrument by clicking on the *Questionnaire* radio button and set the appropriate outcome from there.)
- The *Call as soon as possible* option is always on the dial screen for supervisors and will direct the form to the first available interviewer. The form receives the highest priority (*super*) when this is done.
- *For whom* is used to select the interviewer or group to whom a form should be routed. This button is active only for the dial results *Appointment* and *Call as soon as possible.*
- Edit any fields for which editing is allowed (as set in the CATI Specification Program). If a field can be edited, an asterisk (*) appears in the column next to the field in the *Questionnaire data* section. Select the field and then click the *Edit* button.
- View the *Case summary* and *More Information* dialog boxes from this screen by clicking the *Zoom* button.

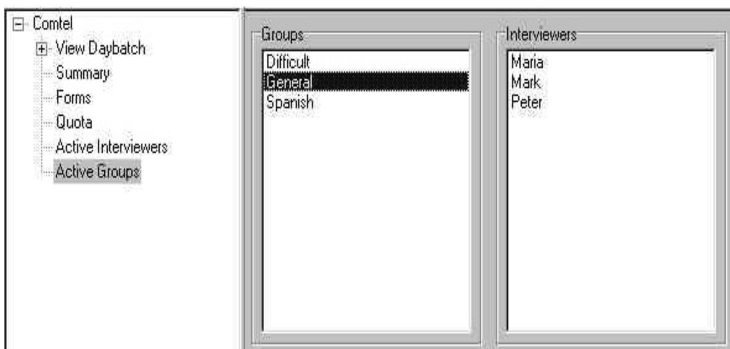
When you are finished, click the *OK* button.

10.5.4 View active interviewers and groups

The option *Active interviewers* allows you to see which interviewers in which groups are currently interviewing.

Click the *Active Interviewers* or *Active Groups* option on the tree and lists of the interviewers and groups appear on the panel. You will see only interviewers or groups that have been defined in the CATI specification file. The following sample shows the *Active Groups* option:

Figure 10–55: View active groups

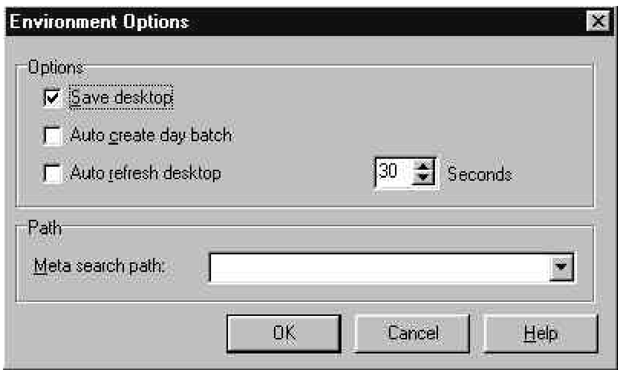


10.5.5 Set environment options

You can set environment options for the CATI Management Program.

Select *Tools* ➢ *Environment Options* from the menu. The *Environment Options* dialog box appears:

Figure 10-56: CATI Management Program Environment Options



Select the settings as described in the following bullets.

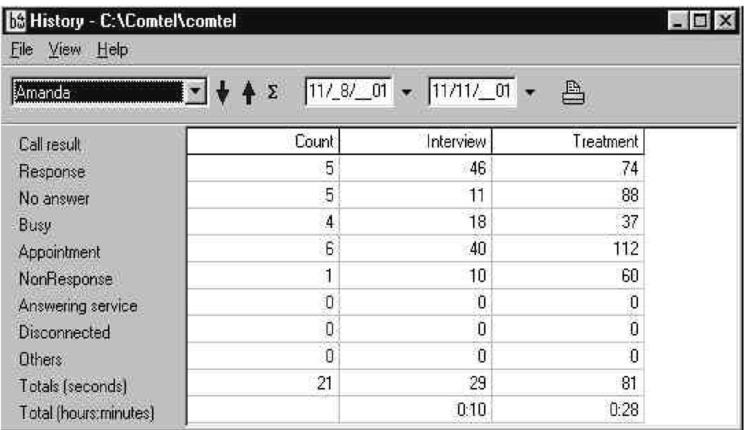
- *Save desktop*: Select to save the desktop in the registry. This option will restore the contents of the desktop when restarting the CATI Management Program.
- *Auto create day batch:* Select to automatically create the daybatch when a specification file is opened, if no current daybatch exists.
- *Auto refresh desktop*: Specify how often (in seconds) the system should refresh the desktop with the latest information from the survey.
- *Meta search path:* Type the path that the system should use to search for the survey's meta information file (`.bmi` extension).

10.5.6 View history and log files

History file

Blaise CATI maintains a history file that contains a record for each attempt (dial) made. The history file is an ASCII file and has the `.bth` extension, in this example `Comtel.bth`. You can view the history file through the *Tools/CATI History Browser* menu option. Two views of the history file are available, a detail view and a list view.

Figure 10-57: History viewer, detail view



Call result	Count	Interview	Treatment
Response	5	46	74
No answer	5	11	88
Busy	4	18	37
Appointment	6	40	112
NonResponse	1	10	60
Answering service	0	0	0
Disconnected	0	0	0
Others	0	0	0
Totals (seconds)	21	29	81
Total (hours:minutes)		0:10	0:28

This *Count* column shows how many of each dial result an interviewer or group of interviewers achieved. Scroll through the interviewer/group list by clicking the down arrow next to the interviewer's name, or clicking the arrows.

The *Interview* column shows the average number of seconds actually spent interviewing; the *Treatment* column shows the average number of seconds spent working on the form, including time spent reviewing the dial screen.

You can look at different days of the survey by clicking the small arrows to the right of the date boxes. A calendar appears, allowing you to choose a different date or range of dates.

If you press the sigma button you will see a summary of that interviewer's (or group's) performance compared to the whole crew over the dates chosen. An example of this summary is given in the following figure.

Figure 10-58: History viewer, detail view

History - C:\Comtel\comtel

File View Help

Amanda | 11/_8/__01 | 11/11/__01

Call result	Count		Interview		Treatment	
Response	(12)	5	(31)	46	(84)	74
No answer	(21)	5	(13)	11	(52)	88
Busy	(29)	4	(10)	18	(55)	37
Appointment	(26)	6	(41)	40	(136)	112
NonResponse	(5)	1	(10)	10	(62)	60
Answering service	(3)	0	(10)	0	(28)	0
Disconnected	(3)	0	(11)	0	(36)	0
Others	(2)	0	(25)	0	(53)	0
Totals (seconds)	(101)	21	(21)	29	(78)	81
Total (hours:minutes)			(0:36)	0:10	(2:11)	0:28

To view the history by list, select *View ➤ List view* from the menu.

Figure 10-59: History viewer list view

History - C:\Comtel\comtel

File View Help

ThePrimKey	InternalKey	DialDate	DialTime	CallNumber	DialNumber
101030 1 3	1	20011108	18:17:53	1	1
101030 2 1	2	20011108	18:18:39	1	1
101030 3 4	3	20011108	18:19:19	1	1
101030 4 3	4	20011108	18:19:46	1	1
101030 5 1	5	20011108	18:20:19	1	1
101030 6 5	6	20011108	18:21:52	1	1
101030 7 2	7	20011108	18:22:21	1	1
101030 8 3	8	20011108	18:22:38	1	1
101030 9 2	9	20011108	18:23:19	1	1
101030 3 4	3	20011108	18:25:55	1	1

1:103

The list view displays a chronological listing of all attempts (dials). You can scroll to the right to see all fields that are included in the history file by default (but not ones you have added).

To create your own summaries of the history file you can use the description of the history file provided in Chapter 11. If you want to do this during production you should copy the history file to another location and create the summaries from the copy. This history file copy should be done with a copying DLL called *blfcopya.dll* that can be supplied by the developers along with 2 Maniplus programs; one that documents how to use it and a second that allows you to test it on your LAN.

View log file

The log file tracks system-level events such as when a daybatch is created, when interviewers log on and off, and so on. It is an ASCII file with a `.log` extension that can be viewed in any text editor.

The following is a sample log file in the Blaise text editor:

Figure 10-60: Sample log file

Comtel.log

comtel.bla | Comtel.log

```
20011108;18:16:23;CatiMana;Mark;Day batch created for 11/8/2001 (1000,0,0,0)
20011108;18:16:36;Data Entry;Mark;Session start
20011108;18:16:49;Data Entry;Mark;Session end
20011108;18:17:52;Data Entry;Mark;Session start
20011108;19:04:52;Data Entry;Mark;Session end
20011108;20:10:35;Data Entry;Mark;Session start
20011108;20:12:16;Data Entry;Mark;No number available
20011108;20:12:50;Data Entry;Mark;No number available
20011108;20:12:55;Data Entry;Mark;Session end
20011108;21:00:39;Data Entry;Mark;Session start
20011108;21:20:31;Data Entry;Mark;Day batch exhausted
20011108;21:20:34;Data Entry;Mark;Session end
20011108;21:22:54;CatiMana;Mark;Treat form (1630,14)
20011108;21:26:00;Data Entry;Mark;Session start
20011108;21:26:12;Data Entry;Mark;No number available
20011108;21:26:14;Data Entry;Mark;Session end
20011108;21:49:40;Data Entry;Mark;Session start
20011108;21:50:09;Data Entry;Mark;Day batch exhausted
20011108;21:50:11;Data Entry;Mark;Session end
20011109;07:21:05;CatiSpec;Mark;Survey definition written
20011109;07:21:34;CatiMana;Mark;Day batch created for 11/9/2001 (996,1,2,1)
20011109;13:05:36;Data Entry;Mark;Session start
20011109;13:34:24;Data Entry;Mark;Session end
20011110;08:56:31;CatiMana;Mark;Day batch created for 11/10/2001 (997,1,1,1)
```

10.5.7 Configure the Tools menu

You can configure the *Tools* menu to run other programs from within the CATI Management Program. When you do this, an option for that program appears in the *Tools* menu. You can run Blaise programs as well as non-Blaise programs. See Chapter 2 for information on configuring tools in the Control Centre, as the procedure is exactly the same.

10.5.8 Running the CATI Management Program outside the Control Centre

You can run the CATI Management Program as a separate program using the `btmana.exe` program file. Use the following syntax:

```
BTMANA SurveyFileName [options]
```

Various program options such as the batch creation of the daybatch, creating a daybatch for the next survey day, the working folder, and others can be set on the command line. See Appendix A "CATI Management Program (btmana.exe)."

The name of the survey file is the same as the name of the data file it belongs to. The extension `.bts` is assumed for the survey file name; if you give it a different extension, the system will still assume that the extension is `.bts`.

10.6 Example: A Simple CATI Survey

To help you see how a Blaise CATI system works, we have provided the example survey `comtel.bla` in the `\Doc\Chapter10` of the Blaise system folder. It has the ingredients needed to demonstrate all parts of the CATI Management System. A Manipula program called `readin.man` will read in a fictitious data set of names and telephone numbers to initialise the data file.

10.6.1 Step 1: CATI data model

Selected parts of the data model of our Comtel example are shown in the following examples:

```
DATAMODEL ComTel "National Commuter CATI Survey, individual follow-up."

  PRIMARY
    Ident, PersonNo

  SECONDARY
    Telephone = Phone
    Form_Stat = Manage.FormStat
    For_Whom = ForWhom
    LastDialResult =
      CatiMana.CatiCall.RegsCalls[1].DialResult

  PARALLEL

    NonResponse = NonResp
    Appointment = Appoint
    OtherOutcome
    NoAnswer
    Answering_Machine = AMachine
    Disconnect
    BusyCall
INHERIT CATI

  INCLUDE "Mylib.lib"

  LOCALS
    LANAME : STRING[30]

INCLUDE "IDENT.INC"
FIELDS
    PersonNo : 1..20
    Phone : STRING[12]
    Wave : 1..20
    KindOfCase : STRING[9]
    TimeZone :  STRING[3]
    TimeSlice : STRING[3]
    Name :      STRING[33]
    ForWhom  :  STRING[10]
    IntName :   STRING[10]
    IntID :     STRING[5]

    CatiSelect: STRING[5]  {This can be any value. It can be  computed from any
criteria in the  Readin.Man program or another Manipula setup.}
    CatiSort  : STRING[5]  {This can be any value. It can be  computed from any
criteria in the  Readin.Man program or another Manipula setup.}

INCLUDE "MANAGE.INC"

{Subject matter blocks below.}

INCLUDE "ADDRESS.INC"   {Address}
INCLUDE "BPERSONT.INC"  {Person}
INCLUDE "WORKPLCY.INC"  {WorkPolicy, called in BWorkT.INC}
INCLUDE "BWORKT.INC"    {Work}

{Parallel CATI outcome blocks below.}
INCLUDE "NONRESP.INC"   {NonResponse}
INCLUDE "APPOINT.INC"   {Appoint}

INCLUDE "DISCONCT.INC"  {For disconnect outcomes}
INCLUDE "AMACHINE.INC"  {For answer machine/service outcomes}
INCLUDE "NOANSWER.INC"  {For no answer call outcomes}
INCLUDE "BUSY.INC"      {For busy call outcomes}
INCLUDE "OTHER.INC"
…
…ENDMODEL
```

Appointment block

A data model for a CATI survey must always contain a user-defined APPOINTMENT block. Since we are using the default appointment block provided by Blaise (the CATI Specification file setting *Disable appointment dialog in the DEP* is not checked), this block contains the questions you would like to ask after an appointment has been made. Making the appointment itself is handled by a special module in the *TAppMana* block. The appointment block you specify is a subsidiary block that collects additional information that may be helpful for other interviewers to know later on.

The appointment-block must be a parallel block specified in the SETTINGS section at the top of the data model. If you assign the name *Appointment* to this block field in the SETTINGS section, the Blaise system will know that it must be used for the appointment treatment.

Inherit CATI

The data model contains the special setting INHERIT CATI. This causes an extra block to be added to the data model. This block contains information used by the CATI Management System. You do not have to worry about what information this special block contains. Everything is taken care of automatically. Just remember that your data model is larger than you think it is.

Telephone number

The telephone number must be included in the data model. In our example, we use the field *Phone* for this purpose. The field should be a text field of sufficient length to store the longest numbers. If you are using the modem feature, make sure the telephone number contains all digits necessary to obtain an outside telephone line.

You are now ready to prepare the data model in the Control Centre.

10.6.2 Step 2: Initialising the data file

Use Manipula to read information from an ASCII file with telephone numbers and other information into the Blaise data set. For our example, we have included the ASCII file `comtel.asc` in the Blaise system distribution.

The file contains made-up names, streets, towns, and telephone numbers. It is set up in a way that will allow you to experiment with the different parts of the call scheduler. This information must be stored in the corresponding fields in various parts of the data model according to the needs of the survey. The Manipula setup

to read the information into the Blaise data file is partially shown in the following example:

```
USES
  CATIForm 'Comtel'

  DATAMODEL InPut  {dat file}
      BLOCK Bident
        FIELDS
          Region "@Yregion Code.@Y" : 0..97
          Stratum "@Ystratum Code.@Y" : 0..9997
          SampleNum "@YSample number.@Y" : 1000..9000
      ENDBLOCK
    FIELDS
      Ident : Bident
      PersonNo : 1..20
      DUMMY[1]
      Phone    :  STRING[15]
      DUMMY[1]
      Name     :  STRING[10]
      DUMMY[1]
      PAddress :  STRING[17]
      DUMMY[1]
      City     :  STRING[11]
      DUMMY[1]
      State    :  STRING[2]
      DUMMY[1]
      TimeZone :  STRING[3]
      DUMMY[1]
      TimeSlice : STRING[3]
      DUMMY[1]
      ForWhom :   STRING[10]

  ENDMODEL

INPUTFILE
  InFile : InPut ('Comtel.asc', ASCII)

OUTPUTFILE
  OutFile : CATIFORM ('Comtel', BLAISE)
```

The USES section of the setup introduces two data models. The identifier COMTEL refers to the data model of the CATI survey, and the identifier INPUT denotes the data model describing the ASCII file. According to the INPUT section, Manipula expects to find the telephone and address information in the file `comtel.asc`. According to the OUTPUT section, the information is written to the Blaise data file `comtel.bdb`.

When you run this Manipula setup, the administrative fields of the data file are filled in, including the time zone, to whom, and time slice fields. Name and address information is also provided, all other fields are empty. You can look at the data file using the Database Browser.

10.6.3 Step 3: Survey specification

Now that you have completed your preparations, you can create a CATI specification file using the CATI Specification Program. If you were doing this from scratch, you would start the CATI Specification Program and open your data model's meta information file with a `.bmi` extension.

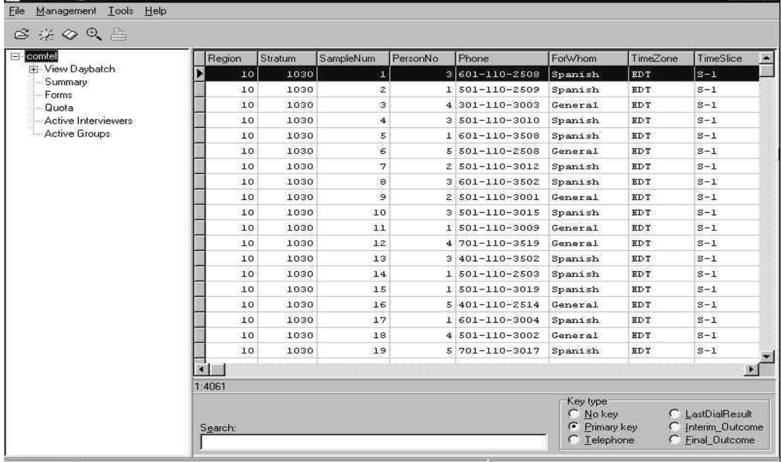
For this example we have provided a sample specification file called `comtel.bts` for you to look at. Previous sections document how to fill in each of the dialogs. You can use a previously created CATI specification file for a new survey in order to avoid having to fill in all parameters each time. In the *Survey days* calendar you can use *Ctrl-C* to clear previous survey days. Interviewer, group, and quota information can be loaded from external ASCII files as documented above.

You would then save the specification file using the menu options. In this example, we have saved our file as `comtel.bts.` Once the specification file is complete, you can move on to survey management.

10.6.4 Step 4: Survey management

The supervisor has to create a daybatch for each day in the survey on which interviewing takes place. This is done in the CATI Management Program. The following figure shows the file `comtel.bts` file from our example:

Figure 10-61: CATI Management Program for Comtel example



To create a daybatch for a new interviewing day in the survey period, select *Management ➤ Create Daybatch* from the menu. The system creates the daybatch and, when finished, a dialog box appears with the results of the daybatch:

Figure 10-62: Create Day batch Results for Comtel example

Create Day Batch - Results

Status	Count
Default	986
Soft appointment	7
Medium appointment	4
Hard appointment	3

OK

Help

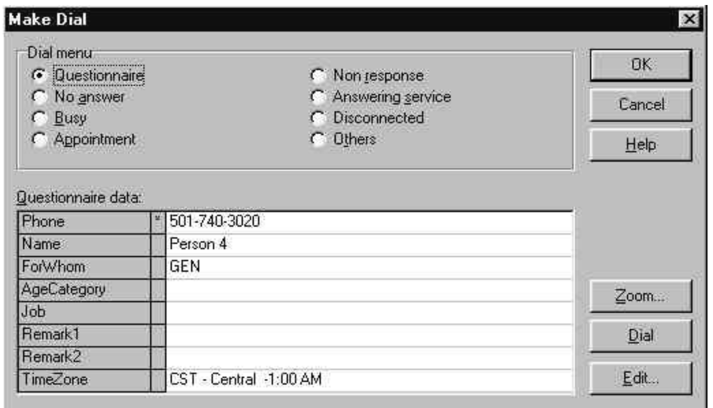
This box shows how many soft, medium, and hard appointments are set for the current day. Click the *OK* button to close the dialog box.

There are many functions in the CATI Management Program, and these are covered in previous sections in this chapter. For the sake of our example, creating the daybatch is the most important step, because without the daybatch you cannot interview.

10.6.5 Step 5: Interviewing

After you have created the specification file and the daybatch, interviewing can start. Run the DEP using the Comtel data model. Each time the system is ready for a new interview, the dial screen appears.

Figure 10-63: Dial screen for Comtel example

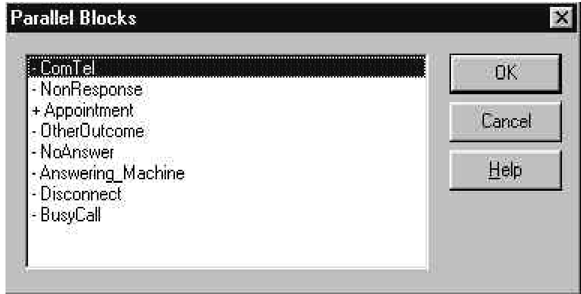


Under the *Questionnaire data* section is the list of selected fields and their values. Of course, the telephone field is one of them. It is always the first in the list.

If the interviewer selects *Start interview,* the DEP will open a form on the screen, and the interviewer can start asking questions.

You can always interrupt the interview when necessary and access the appropriate parallel blocks. This can be done by selecting *Navigate ➤ Sub Forms* from the menu. Our Comtel example has the following parallel blocks:

Figure 10-64: Parallel blocks for Comtel example



The interviewer selects the appropriate option and clicks the *OK* button. If she chooses the *Appointment* block, the *Make Appointment* dialog box appears. (This is described earlier in this chapter.)

An interviewer may also interrupt the interview by using the key combination Ctrl-Shift-Home to return to the dial screen.

This completes our discussion of our CATI example.

10.7 CATI/CAPI Compatibility

If you are using your survey for Computer Assisted Personal Interviewing (CAPI) as well, with interviewers using laptop computers to conduct field work, you can use the same data model for CAPI as for CATI. This will ensure that the CAPI data model has the same data definition as the CATI data model. But there are a few considerations.

Override CATI mode

You can run the CAPI instrument with the */D* parameter to override the CATI mode. This means that the CATI Call Management System will not come up on the laptop, which is a different interviewing medium that may have its own management criteria. For example, a field interviewer will often organise her work around the geography of the respondents, no matter which surveys she is working on at the moment.

CAPI appointment block

If you disable the CATI mode with */D*, you can still use the appointment-making module of the CATI system, but only if the CATI specification file (`.bts` extension) is present. If you do not have this file present, and still want the ability to make appointments, you must insert a completely separate appointment block that is invoked only in CAPI mode.

Manually access forms

Another option is to allow the CAPI interviewer to use the CATI system to manually access forms while in CATI mode. She can use the menu options *New, Get,* or *Browse,* as well as the CATI dial screen.

Using the call scheduler

In CAPI, you do not have the power of the networked environment for survey management, handling appointments or busy signals, or accumulating quota counts. On the other hand, you can use the appointment-making facility of the call scheduler to keep track of appointments, busy signals, and the like, but only for the interviewer on the laptop.

10.8 Other Considerations

Dial result field

As a survey goes on for many days, you usually have to read data into or out of the CATI data file. In the Comtel example, there is a field in one of the provided CATI management blocks known as the *DialResult* field. It is declared as a secondary key. When it is time to read completed forms out of the data set, you can use a Manipula program to key on a value of *DialResult*. See Chapter 8 for information on reading just those forms with a complete value for *DialResult*. The CATI Call Management System updates the value of *DialResult* automatically.

Manipula on a live data set

You can run Manipula while a Blaise data set is in use using ACCESS=SHARED. With this you can read forms into or out of the data set, or make customised reports without interrupting the calling. You have the option of skipping forms currently in use, or having Manipula wait while the in-use forms are cleared. Running Manipula with ACCESS=SHARED is very slow due to the security measures taken to protect the data set. A way to minimise this slow performance is to run a Manipula program once against the data set, extracting enough information that one time to into an ASCII file and run a number of reports from that ASCII file. An alternative way to avoid a long Manipula run is to write extra survey management information to a history file and use the elaborated history file as a source of real-time reports and listings (see above on how to access this history file during production).

CATI testing utility (CATI Emulator)

A CATI testing utility, `btemula.exe`, is included with the Blaise system software. With this utility you can emulate an interviewing session without interviewers. You can test the call scheduler, or test the load on the LAN made by numerous machines operating at one time. This program can work off of a script of data values and fill in forms in the data set.

Modify daybatch

Maniplus, an extension of Manipula, allows you to modify the daybatch without recreating it. You can add a form to or remove a form from the daybatch during production. See the Maniplus manual for more information.

Resetting some management fields upon re-entry

When you re-enter a not-completed form a second time it can be useful to reset the contents of some management fields or blocks to empty. For example, if an *Other* outcome had been recorded and fields in the *Other* parallel block filled in during the previous attempt, you would want the *Other* parallel block emptied before another attempt is made so the interviewer doesn't see the previously entered values for this management data. In the Comtel.bla file is an auxfield called *ReEntry*. When this auxfield is empty, as it will be upon re-entry, some blocks and fields are emptied out. Then the auxfield *ReEntry* is populated with the value *Yes*. Thus the emptying out of these management fields and blocks is done only once per session.

Backing up and archiving

A typical CATI survey lasts anywhere from several to many days. A common practice is to run jobs overnight in batch after the calling day. These jobs may include reading forms into or out of the Blaise data set, creating reports, updating records, and so forth. A concern with any data set (Blaise or not) is that over time it may become corrupted or too large. A way to avoid large problems is to handle them when they're small. If you run Hospital/Recover and a Blaise-to-Blaise Manipula program every night, you can root out small data problems, and compact the data set, as part of normal operations. Another helpful practice is to archive the data set before every major batch job in a compressed file. This way if the batch processing is interrupted (say by a power failure) you can pick it up where it left off instead of having to run the whole thing when you first arrive at the office. Additionally if there are problems with the batch processing, having snapshots of the data set at each major step can be an aid in diagnosing problems.

Controlling the call scheduler

Broadly speaking, there are two ways to control the execution of the call scheduler. First is to control the composition of the daybatch. Second, once a daybatch is created, is to control the delivery of forms from the daybatch. Both aspects of controlling the call scheduler are equally important. It is possible to create reports from Manipula that indicate for example, whether your study is cycling through the same forms day after day while leaving others untouched. You should continually monitor the creation of the daybatch and the delivery of forms from the daybatch during calling.

Changing the CATI specification file settings during the course of a survey

The CATI Specification Program settings should be considered to be dynamic and changeable as a survey progresses. Settings that are appropriate for the start of a survey, where you are trying to touch all cases as fast as possible, may not be

suitable for the end of a survey where you're trying to reach relatively few cases. It is useful to think of three phases of a survey: Start up, Cruising, and Finish. For each survey phase, consider the following guidelines (not a complete list):

- *Start up:* Enable time slices, low maximum number of dials, and moderate number of busy dials.
- *Cruising:* Time slices still enabled, but with finer divisions (2nd time slice set), moderate maximum number of dials, and moderate number of busy dials.
- *Finish:* Disable time slices, high number of maximum dials, high number of maximum busy dials.

Windows® NT setting

Blaise CATI works well in a variety of network environments. However, under Windows® NT there is a setting you must be aware of. The setting, *opportunistic locking*, must be turned off. A network administrator must do this. If this setting, controlling a method of buffering data, is not turned off you will corrupt the data file. It is not a matter of “if,” it is a matter of “when.”

11 CATI Technical Details

This chapter describes some technical aspects of the CATI Call Management System. You do not need to know the material in this chapter to effectively use the CATI Call Management System. All the information you need to know is included in Chapter 10. This chapter is for users who want to know more about how this system works.

In this chapter we will discuss:

- Rules for inclusion in the daybatch.
- The operation of the call scheduler, including selecting forms, routing back forms, and assigning priorities.
- Treatment types, treatment of dials, and exceptions to treatment rules.
- The files needed for a CATI survey.
- The history file.

In the last section of this chapter, a glossary of CATI terminology is provided.

11.1 Rules for Inclusion in the Daybatch

A form is a candidate for inclusion in the daybatch if it meets the following conditions:

- The form has not been concluded yet, meaning that the last call has yielded one of the dial results *nonresponse, response, disconnected*, or *other*.
- There is no appointment for the form, or a previous appointment for the form has expired. The telephone number has not yet had the maximum number of calls.
- The form has an appointment that is current and that can be met on the interview day. For the various types of appointments, this means:

Hard appointment. The agreed date for the appointment has been reached, or today is the first interview day after the agreed date.

Preference for a period with day part. Today is within the specified period, and there will be interviewing activity on the specified part of the day.

Preference for a period without day part. Today is within the specified period.

Preference for a day in the week with day part. Today is one of the specified days of the week, and there will be interviewing activity on the specified day part.

Preference for a day in the week without day part. Today is one of the specified days of the week.

Preference for a day part. There will be interviewing activity today on the specified day part, or today is the last day of the survey period.

- The telephone number field is filled in.
- If *Daybatch select* fields have been defined in the CATI specification file, the conditions specified for inclusion are satisfied, or the conditions for exclusion are not satisfied.
- If time slices are used, a form that does not have an applicable appointment for the current day but has already been attempted at least once on a previous day is included in the daybatch only if there is a time slice available that is current.

The forms are ordered according to their appointment types. The order is:

- Hard appointment.
- Preference for a period with day part.
- Preference for a day in the week with day part.
- Preference for a period or for a day in the week without day part. These two cases are treated together for the assignment of a starting time to the form.
- Preference for a day part.
- No appointment/preference or expired appointment/preference.

For each of the above-mentioned groups of forms with preference, the forms with last possibility are put at the start. (The term *last possibility* applies to forms for which an appointment with preference has been made, and it is at the end of the period to contact them.) Last-chance situations are identified independently of the day part.

After sorting the forms according to the appointment type and last chance, the program will further sort the subgroups:

- If *Daybatch sort* fields have been defined in the CATI specification file, it sorts the forms according to the specification.
- If *Daybatch sort* fields have not been defined, it sorts the forms according to the number of previous calls. It then shuffles the numbers within the new groups at random.

The forms will be included in the daybatch from the sorted list, until either the daybatch is full or all forms have been included.

11.2 Call Scheduler

The call scheduler is responsible for scheduling the telephone numbers in the daybatch, making the forms available at the right time. The scheduler can also modify a form's priority, if necessary. A large number of parameters can be set to control the behaviour of the scheduler.

The scheduler is automatically activated by the Data Entry Program of the interviewer who is the first to ask for a new form in a certain time interval, or when an interviewer concludes a call after a time interval has elapsed. Usually, scheduling takes less than a second.

Statuses in the daybatch

The scheduler works by assigning one of the following statuses to forms in the daybatch:

- *Being treated.* A dial is currently being performed for this number.
- *Busy.* The result of the last dial was busy and the form will be given the busy treatment.

- *No-answer.* The result of the last dial was no-answer, and the form will be given the no-answer treatment.
- *New Appointment for Today.* An appointment for today has just been made for this form. The next schedule will determine whether the new status should be active or not-active.
- *No need today.* This form should not get another dial in the current daybatch if the dial result is final.
- *Not-active.* The form cannot get a dial at the moment. The form will be given the status *active* at a certain time.
- *Active.* The form is ready for a dial.

The treatment of a telephone number in the daybatch depends on its status.

Using information from the daybatch

The scheduler uses the information in the daybatch. This information includes:

- The interval in which a number should be activated (start interval).
- The interval after which the number should no longer be active (end interval).
- The time difference between the respondent and the interviewer.
- The priority that will be assigned to the number at the moment of activation (future priority).
- The number of dials within the current call (dials).
- The interval in which the number was dialled last (dial interval).
- The number of times in a row that the telephone number has had the status *busy* (busy dials).
- Quota information.
- Time slice information.

For every form with the status *not-active*, the scheduler checks whether its start interval has been reached. If it has, the status of the number becomes active, and it is assigned a priority based on its future priority.

11.2.1 Selecting forms

When an interviewer requests a form from the dial screen, a form is selected from the daybatch. This selection is performed as follows.

The system first determines whether there are active forms. If there are active forms, the first form in the daybatch satisfying the following three conditions will be selected:

- It has the highest available priority. Only forms that can be treated by the interviewer asking for a form are taken into account.
- It is the most suited form. The following ordering is used to determine this, in decreasing order of being suited:

  a) The form is for this particular interviewer only, including already expired forms.

  b) The form is for the main group of the interviewer, including already expired forms.

  c) The form is for one of the other groups to which the interviewer belongs, including already expired forms.

  d) The form is for everyone (not for any specific interviewer or group), or it is an expired form.
- It is the most urgent form. The form has the lowest number of dials within the daybatch. If there are other forms with the lowest number of dials, it is the form that has been active for the longest period of time.

The second condition is applied only if there is more than one form satisfying the first condition. The third condition is applied only if there is more than one form satisfying the second condition. Note that even if a form has expired, the interviewer or group to whom it was originally meant to go still has priority over the other interviewers or groups.

The telephone number that has been selected for an interviewer is given the status *being treated*.

11.2.2 Routing back forms

In order to have the scheduler route a form back to a certain interviewer or group of interviewers, you have to do two things.

- First, define a route-back (or *ToWhom)* field in the data model. The *ToWhom* field specifies the destination the scheduler must use. The *ToWhom* function is one of the functions that can be assigned to a field in the CATI specification file.

- Second, define the destinations by specifying which groups and interviewers will be working on the survey. This is done in the CATI specification file. The name for an interviewer is determined by the system, first by referencing the registry of the interviewer's workstation. The value stored for BlaiseUser in the environment subfolder of the HKEY_CURRENT_USER subfolder is assumed to the interviewer's name. If this value is not present, the system uses the login name to identify the interviewer. An interviewer can be a member of more than one group.

While building a daybatch, the system determines which numbers have to be routed to a specific destination based on the value of the *ToWhom* field. This destination can be either an interviewer or a group. If the *ToWhom* field is empty, the case can be routed to any interviewer.

If an appointment is made by an interviewer, the contents of the *ToWhom* field will be changed according to the value of the route-back parameter, as set in the CATI specification file:

- If you select to route back to *Interviewer,* the *ToWhom* field will be filled with the name of the interviewer who made the appointment.
- If you select to route back to *Group*, the *ToWhom* field will be filled with the name of the main group the interviewer belongs to. If the interviewer belongs to no group, the field will be made empty.
- If you select *Do not change route back*, the contents of the field will not be changed.

During production, the supervisor can change the route-back information by selecting the form for further treatment. (The procedure for this is described in Chapter 10.) By changing the *For whom* box on the *Treat Form* dialog , the form can be routed to a different group or interviewer. The *ToWhom* field will then be filled with this value. If the option *Everyone* is selected, the *ToWhom* field will be made empty.

11.2.3 Assigning priorities, starting times, and ending times

When the scheduler is activated, it assigns a priority to each form in the daybatch. The scheduler determines which forms to select on the basis of their priority. The following table summarises the priorities.

Figure 11-1: Priorities

Situation	Priority	
Appointment made by supervisor	8	Super
Hard appointment for date and time, busy	7	Hard-busy
Hard appointment for date and time	6	Hard
Appointment with preference, last possibility, busy	5	Medium-busy
Appointment with preference, busy	4	Soft-busy
No appointment, busy	3	Default-busy
Appointment with preference, last possibility	2	Medium
Appointment with preference	1	Soft
No appointment	0	Default

The *Appointment made by supervisor* is the result of a *Call as soon as possible* dial made from the supervisor's dial screen.

Last-chance forms with a preference appointment are always given medium priority. Other forms with a preference always get soft priority. Forms with an appointment for date and time get hard priority. Forms without appointment/preference and with expired appointment/preference get the default priority. A form's priority is only relevant when its status is *active*.

The soft, medium, and hard priorities are always the result of appointments with preference or of hard appointments.

Priorities are adjusted during scheduling if necessary. For not-active forms, the system determines which priority they will get on activation. This priority is called the *future priority*.

If a hard appointment expires before it is met, it will be assigned medium priority on the first survey day after the day for which the appointment was made. On the other days it will get default priority. Expired preferences get default priority.

Forms with day part preference get medium priority on the last day of the survey. On the other days, they will get soft priority.

Start and end times

The starting time for forms with hard priority is the time for which the appointment was made. The ending time for such forms is not relevant.

Forms with default priority for which the time slice mechanism is not used will be assigned the starting time specified in the CATI specification file for the starting time of the first crew. The scheduler cannot deliver these forms before this time. If time slices apply, then the start and end times for the form will be set to the start and end times of the first available time slice definition.

If a form with preference for a period or for a day in the week without day part has soft or medium priority, it is assigned a starting time based on the day's crew definition. The numbers are distributed over the various crews in accordance with crew capacities. The starting time of a number is the starting time of the crew it is assigned to; its ending time is the ending time of the last crew. The capacity of a crew is defined as the number of interviewers multiplied by the number of intervals in which the crew is active.

The starting and ending times of all other numbers with preference are derived from the specified day part. Medium priority numbers will be active for at least one hour. This may require adjusting the starting and ending times.

The starting and ending times are corrected for the *Do not call before* and *Do not call after* settings.

Time slices and the call scheduler

The call scheduler takes time slice information into account, and distinguishes between two situations.

- In the first situation, the number did not receive a dial between the start and end time set for that number. Therefore, the number will be de-activated. The system determines whether the form has an untried slice definition for the same day. If so, the number will receive a new start and end time and the *not-active* status. It will be activated when the new start time is reached. If no time slice is available, the number receives the *no need today* status.
- In the second situation, the number received a no-answer dial. The no-answer treatment will be given. See the section 11.3 for information about Treatments.

11.2.4 Activating a form with medium or higher priority

When activating a form with a medium or higher priority, the scheduler determines which interviewers are currently working on the survey. Now there are two possibilities: the case has to be routed back to an interviewer or routed back to a group. Remember that the following description applies only to cases with medium or higher priority.

Form routed to an interviewer

If the form has to be routed to an interviewer, the case is activated for the specified interviewer. The scheduler will wait for him until the *Interviewer de-activation delay* (as specified in the *Scheduler parameters* in the CATI Specification Program) has elapsed. Again there are two possibilities:

- The interviewer belongs to no group. In this case, the form will be routed to any interviewer currently working.
- The interviewer belongs to a specified group. In this case, the scheduler will try to route the form to someone in that group. The scheduler will wait for one of them until the *Group de-activation delay* has elapsed. If a member of the specified group is still unavailable by that time, the scheduler will expire this route. The form can then be routed to any interviewer.

Form routed to a group

If a form has to be routed directly to a group, the form is activated for the specified group. The scheduler will wait for a member of the group until the *Group de-activation delay* has elapsed. If a member of the specified group is still unavailable by that time, the scheduler will expire this route. The case can then be routed to any interviewer.

Expire on de-activation delays only

If *Expire on de-activation delays only* has not been checked in the *Scheduler Parameters* of the CATI specification file), the interviewer de-activation delay is applied only if the specified interviewer is on the system. This means that if the specified interviewer is not running the data entry program at the starting time for the relevant appointment, the form can be routed to anyone. Likewise at least one member of the target group needs to be on the system for the group de-activation delay to be applied. If *Expire on de-activation delays only* has been checked in the CATI specification file, the interviewer and group de-activation delays are applied irrespective of who is and is not on the system.

If a form has expired because a specific interviewer or group was not available, and that interviewer or group is available in the next schedule, that interviewer or group will still have priority over other interviewers or groups.

An example

Let's look at an example. Suppose you have a case with a hard appointment for 16:00 hours for interviewer Jones. So the value of the *ToWhom* field is *Jones*. *Expire on de-activation delays only* has not been checked in the CATI specification file. Interviewer Jones is not working on the survey when the scheduler activates the case and so the interviewer de-activation delay does not apply. If Jones has been assigned to the group *Experienced* and an interviewer of that group is available, the form will be activated for all members of that group. If Jones has not been assigned to a specific group, the form is activated for all interviewers.

As another example, suppose you have a case with no preference for the group *French speaking*. Because the priority of such a form is less than medium, the form will only be available to members of the specified group.

The interviewer and group de-activation delays mentioned above can be specified separately for medium priority and for hard or higher priority. Note that the de-activation delays apply only to forms with medium or higher priority.

Let's look again at the Jones example. Suppose you have a hard appointment for the 17:00 hour, but interviewer Jones doesn't log onto the system until 16:30. His first interview is very long. The *Interviewer de-activation delay* is 15 minutes and the *Group de-activation delay* is 30 minutes. At 17:10, Jones asks for a new form. The form with the 17:00 appointment is still available for Jones, because the interviewer de-activation delay has not elapsed. If, however, Jones doesn't request a new form until 17:30, there is a chance the form is no longer available for him. Another member of the group *Experienced* may already have handled it. If the form has not been handled yet, it is still available for all members of the *Experienced* group only. This is because the group de-activation delay has not elapsed yet.

11.3 Treatments

The treatment a form gets depends on the dial result. The CATI system distinguishes a number of treatment types:

- A no-answer or answering service dial result gets the treatment *no-answer.*
- A busy dial result gets the treatment *busy.*
- An appointment dial result gets the treatment *appointment*.

Forms with the dial results *disconnected*, *other*, *response*, and *nonresponse* are considered concluded, and therefore no special treatment is needed.

For both the no-answer and busy treatments, we talk about dials and calls. As a general rule, all dials for a given telephone number on a given day make up one call. There are two exceptions to this rule, which are described below in section 11.3.2.

No-answer treatment

With the no-answer treatment, a number of dials, up to the limit set in *Maximum number of dials* in the CATI specification file, are performed during the rest of the day.

When this maximum is reached, the telephone number is assigned the *no need today* status. As a result it will no longer be presented to an interviewer in the same daybatch. If the maximum has not yet been reached, the treatment depends on the form's future priority, which becomes its actual priority at activation time:

- If the future priority is *default*, the form will get the status *not-active*. The number can get another dial as soon as the time specified in the *Minimum time between 'other' no-answer* has elapsed. If there is a time slice set and dials are allowed on the same day, the system determines whether there is another slice definition available during the same day that has not yet been tried for this form. That slice definition has to have a start time that is far enough in the future—later than the earliest time that the number can become active again based on the time specified in the *Minimum time between 'other' no-answer*. If so, the number will receive a new start and end time and the *not-active* status. If no slice is available or no other slice may be tried, the number receives the *no need today* status. When and if this will actually happen depends on the size and the composition of the daybatch.

- If the future priority is *hard* then the status will become *not-active*. The number can be phoned back after a number of minutes, according to the *Minimum time between hard/super no-answer* setting.
- In all other cases (future priority *medium* or *soft*), the status will also become *not-active* and reactivation time will have to be calculated. This calculation is carried out as follows: The time that is still available for the telephone number (this depends on the form's dial time and end time) is divided by the number of dials left for the form in the current daybatch. The result is added to the current dial interval.

For example, suppose a number has to be dialled between 13:00 and 17:00 hours and its maximum number of dials is six. The first dial is carried out at 13:30 with result *no-answer*. So the number can be phoned back during 3½ (up to 17:00 hours) and can get five more dials. The next time the number will be dialled is 14:05 = 13:30 + 0.35 (= 3½ hours divided over five dials).

The delay after which the number will be dialled again is calculated to be at least the value specified by the parameter *Minimum time between 'other' no-answer* in the CATI specification file.

Remember that the CATI system always works with 5-minute intervals. For example, if a number has to be phoned back 15 minutes later, it will be activated when three intervals have elapsed. If the number has been dialled at 14:04:59, it will be reactivated at 14:00 (because this is the dial interval) + 0:15 minutes (three intervals) = 14:15. This is less than 15 minutes later! The reactivation time is calculated based on the number of intervals.

If the no answer treatment is being applied to a form with an *answering service* dial result and the setting *Do not allow multiple same day answering machine calls* is checked in the CATI specification file, *no need today* status is assigned to the form. In such a case, the answering service dial result is normally being used to indicate that a message has been left and so the form should not be called again on the same day.

The busy treatment

The idea behind the busy treatment is that a number with the dial result *busy* must be quickly dialled again as there is a fair chance to get somebody on the line. It is also possible that the receiver has not been replaced. So if a number has had the dial result *busy* several times in a row, it is better to stop the busy treatment. You can specify the maximum number of busy dials for the busy treatment. You can also specify the time between subsequent dials.

The scheduler keeps a count of successive busy dials. It uses this count to determine which one of the eight possible *Minutes between busy dials* settings from the CATI specification file should be applied. When and if the limit set in *Maximum number of busy dials* is reached, the scheduler applies the no answer treatment to the busy dial result. This means that the scheduler has given up on chasing a series of busy dials and is choosing to treat lump these busy dials together into the equivalent of one no answer dial.

The appointment treatment

The appointment treatment sees to it that appointments will be dealt with correctly, so that numbers with an appointment will be included in or excluded from the daybatch depending on the appointment information. The appointment information can either be filled in by an interviewer or it can be imported into the Blaise® data file from an ASCII file.

11.3.1 Treatment of dials

Each time an interviewer selects to retrieve a form in the Data Entry Program, the scheduler checks whether there are forms that can be dialled and selects the most appropriate one (see the *Selecting forms* section in this chapter). After selecting the form, the dial screen is presented to the interviewer. If no more forms are available, then an appropriate message is presented to the interviewer.

A number that is busy will be given a number of busy dials (see the explanation of *Busy treatment* above in this section). If there is no answer, or if contact is made with an answering service, the interview cannot be carried out. In this case a number of dials may be performed later in the day (see the explanation of *No-answer treatments* above in this section). If no contact has been established by the end of the day, the number will be contacted another day in the survey period unless it has had the maximum number of calls.

If it appears to be impossible to make contact (for example, because the number does not exist or has been disconnected), no more calls should be made. The interviewer must then choose a proper final dial result for the number, such as *nonresponse* or *disconnected number*.

11.3.2 Exceptions to general treatment rules

There are two exceptions to the general rules for the treatment of forms. For these exceptions, the program ignores the maximum number of calls as specified in the CATI specification file.

The exceptions are:

- If an appointment is made for today, the new appointment is treated as a new call.
- On the very last interview day of the survey period, once the day part has expired for forms with medium priority (soft appointments receive medium priority on the last day), these forms remain in the daybatch with default priority. Any dial attempts made on these forms after the day part has expired are considered part of a new call.

11.4 Files Needed for CATI

This section lists the various files that are needed for a CATI survey. Files marked with an asterisk (*) are optional.

Figure 11-2: Files needed for CATI

General Blaise® Files	
Name	**Function**
<file>.bmi	The meta-information file
<file>.bdb	The Blaise data file
<file>.bdm	Page layout for the Data Entry Program
<file>.bfi	The file-info file
<file>.bjk	The join-key file
<file>.bpk *	The primary key file
<file>.bsk *	The secondary key file
<file>.brd *	The remarks data file
<file>.bri *	The remarks index file
<file>.log *	The log file
<file>.bxi*	Extra meta information

Figure 11-3: Additional files needed for CATI

Specific CATI Files	
Name	**Function**
<file>.btd	The daybatch data file
<file>.bti	The daybatch index file
<file>.bts	The survey definition file
<file>.bth	The history file
<file>.bc	The counts file

You can pre-set basic information for a survey in the CATI specification file (`.bts` extension). Then, after setting the parameters, you can copy the file to other survey names or to other locations. (See Chapter 10 for details on creating a CATI specification file.)

11.5 History File

The history file stores information on the dials made by the interviewers or the supervisor using the data entry program. The history file contains the following information about the number that has been dealt with:

- The primary key
- An internal key
- The date
- The dial time
- The number of calls
- The number of dials
- Interviewer who made the dial
- The priority before the dial was made
- The priority after conclusion of the dial
- The dial result
- The line number in the dial menu that was selected by the user
- The appointment type (if the dial result was *Appointment*)
- The time when the dial was completed
- The time in seconds needed for the complete dial
- The time in seconds needed for the interview

The file can be viewed using the `bthist.exe` program (this is described in Chapter 10). The history file is a delimited file, and the fields are separated by commas. If additional fields from the datamodel have been selected for inclusion in the history file (see section 10.4.6), the values in these fields can be seen by opening the history file (it has a .bth extension) as a regular text file. The fields

are shown in the same order as they are presented in the CATI specification file in the *Field Selection* subbranch.

The following data model, `history.bla`, describes the history file (this can be found in the `\Doc\Chaptr11` folder):

```
DATAMODEL history

  SECONDARY
    InternalKey, DialDate, DialTime

  TYPE
    TEntryPrio = (Default            (1),
                  SoftAppoint        (2),
                  MediumAppoint      (3),
                  BusyDefault        (4),
                  BusySoftAppoint    (5),
                  BusyMediumAppoint  (6),
                  HardAppoint        (7),
                  BusyHardAppoint    (8),
                  SuperAppoint       (9))

    TDialResult = (CResponse         (1),
                   NoAnswer          (2),
                   Busy              (3),
                   Appointment       (4),
                   NonRespons        (5),
                   AnswerService     (6),
                   DisConnect        (7),
                   Others            (8))

    TExitPrio = (Busy                (1),
                 NoAnswer            (2),
                 NewAppointForToday  (3),
                 NoNeedToday         (4))

    TAppointType = (NoPreference     (1),
                    CertainDate      (2),
                    PeriodDaypart    (3),
                    WeekdayDaypart   (4),
                    PeriodOnly       (5),
                    WeekdayOnly      (6),
                    DaypartOnly      (7))


  FIELDS
    {The actual length of the primary key depends on the questionnaire.}
    ThePrimKey : INTEGER[12], 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]

ENDMODEL
```

Using the following Manipula setup, you could convert the history file to a Blaise data format:

```
SETTINGS
 DATEFORMAT = YYMMDD

 USES HISTORY 'history'

 { filename with extension BTH required for the INPUTFILE! }
 INPUTFILE histin: history (ASCII)
 SETTINGS
   SEPARATOR = ','

 OUTPUTFILE histout: history (BLAISE)
```

You can use the Database Browser to look at the file created by this setup. The forms in the file can be displayed in order of the secondary key of the history data model.

11.6 Glossary

The following is a list of the terms that can be of help in understanding CATI.

- *Active day.* An active day in a survey is a day on which interviewing will take place. Active days are highlighted on the various calendars in the program.
- *Appointment type.* CATI distinguishes a number of appointment types. If a respondent specifies a date and a time, we call this an *appointment*. In all other cases, we use the term *preference*. The scheduler uses the various appointment types for calling back respondents. Appointment types are no preference, specific date and time), preference for a certain period with day part, preference for a week day with day part, appointment with preference for period, appointment with preference for a day in the week, and appointment with preference for a day part.
- *Busy dial.* A busy dial is a dial that is performed when the number is busy. A number of busy dials together form one dial. The idea is that a number that is busy should quickly be dialled again, since there is a good chance to make contact, as somebody seems to be on the line. But at some point it is wise to stop dialling after a specified maximum number of busy dials. If this happens, the number of dials for the telephone number is increased, and the system tries to contact the number again later (unless the number has had the maximum number of calls and no appointment has been set). Using busy dial intervals, you can specify how long the scheduler has to wait between consecutive busy dials.

- *Call.* A call is a set of logically related dials. A maximum number of dials can be performed for each call. This maximum is specified in the CATI specification file. In principle, a telephone number can only be subjected to one call on a given day. A number can have more than one call on a day, for instance, if an appointment has been made for the same day. The maximum is ignored if an appointment was made for the number.
- *Counts file.* The counts file contains counts on the status of the forms in the Blaise file. It is used to display the summary and quotas in the CATI Management Program. The file is created when a new daybatch is created, and it is kept up to date by the CATI system. If you want to see the summaries and the file is not present, you will be asked if it should be created. This can be done only if no interviewing is going on. The file extension is `.btc`.
- *Daybatch.* A daybatch is a file that contains only those forms for respondents who can be contacted on a specific day in the survey period. You have to create a daybatch in order to work with CATI.
- *Day part.* A day part is a row of consecutive intervals. Each day part is bounded by two intervals. Internally, parts of the day are specified by the interval numbers with which they begin and end. For instance, the day part from 11:00 hours to 13:00 hours is bounded by intervals 133 (11:00-11:05 hours) and 156 (12:55-13:00 hours).
- *Dial.* A dial is the process of selecting a form and making it available to an interviewer, who tries to get a respondent on the line. After the conclusion of the dial, the form acquires a dial result.
- *Dial result.* A dial result is the result that has been registered for a dial. A dial result determines what treatment should be given to a form selected by an interviewer.
- *Future priority*. The future priority is the priority the number will have at the time the scheduler gives it the *Active* status. This is the start time as stored in the daybatch for this form. The future priority is established after the conclusion of a dial and is stored in the daybatch.
- *History file*. The history file stores information on the dials made by the interviewers and the supervisor. The extension of the file is `.bth` and it is viewed by running the `bthist.exe` program.
- *Interval.* An interval is a unit of 5 minutes. CATI divides the day into 288 intervals (numbered from 1 to 288). Each interval is bounded by two times, its starting time and its ending time. For example, interval 1 is bounded by

00:00 and 00:05, interval 288 by 23:55 and 24:00, and interval 100 by 08:15 and 08:20. In CATI, all time operations are performed in intervals.

- *Last possibility.* The term *last possibility* applies to forms for which an appointment with preference (not a hard appointment) has been made. For numbers with a period appointment, the last possibility is the last day in the appointment period. For numbers with a weekday appointment, the last possibility is the last possible weekday for the appointment in the survey period. For numbers with a day part appointment, the last possibility is the last day in the survey period.
- *Period.* A period is a row of consecutive days in the survey period. The period includes both the active and not-active days.
- *Priority*. The priority of the telephone numbers in a daybatch determines the sequence in which the numbers will be presented to the interviewers. The priority is only of importance for numbers with the active status. CATI assigns a priority to telephone numbers based on the type of appointment that has been made and whether the last dial result was a busy.

High-priority numbers are dealt with first. When an appointment has been set for a specific date and time, its priority is always *hard* on the specific date. The priority is *soft* when an appointment is set with preference for a period, unless it is the last possibility to contact the respondent (in which case the priority is *medium*). If a number has no appointment or has an appointment without preference, or if previous dials have led to the result busy or no-answer, then the priority is *default*. Finally, appointments that have been made by the supervisor have the priority *super*. (See the table in the *Assigning priorities, starting times, and ending times* section of this chapter.)

Scheduling

Scheduling is the method by which forms in the daybatch are made available to be dialled. The part of the program that performs this function is called the scheduler. The scheduler judges the priority and the status of the forms in the daybatch and adjusts them if necessary.

Specification file

The CATI specification file is a file containing settings that define when and how a survey should be executed. This includes, for example, the period of the survey, the days on which interviews will be held, and the number of crews and interviewers. This file also indicates which treatment must be given to telephone numbers in certain situations, such as how many times a number must be called back if it is busy or there is no answer. The specification file is created in the

CATI Specification Program and has a `.bts` extension. You must have this file in order to run the CATI Management Program and your CATI survey.

Status

The status refers to the status of the form in the daybatch. You can see the current status of the telephone numbers in the daybatch in the CATI Management Program. The status of a form can be *being treated, busy, no-answer, appointment, no need today, not-active,* or *active*.

Supervisor

The supervisor is the person who takes care of a survey while interviewing is going on. The supervisor is the one who creates the daybatch, solves problems with the equipment, assists the interviewer with technical or equipment problems, and talks to respondents who want to be contacted immediately and possibly makes appointments with them.

Survey period

The survey period runs from the first to the last day of the survey. You specify the survey period in the CATI specification file.

Time slice

Time slice is a mechanism to spread the dial attempts for default-priority cases over time according to a user-defined scheme, based on days and day parts.

Time zones

A time zone is an area where the time is equal. Large areas or countries are usually divided into a number of time zones, which usually, but not always, differ by at least one hour. For example, the United States spans seven time zones. The Blaise CATI system allows you to store time zone information, and the scheduler takes into account the time difference between the interviewer’s location and the respondent’s location. (See Chapter 10 for details on using time zones in your survey.)

All times in CATI are stored in the respondent’s time. For example, if an interviewer working in Washington, DC calls a respondent in Denver, CO, the time in Denver is two hours earlier than the time in Washington. If the interviewer makes an appointment for 16:00 hours (4:00 p.m.), then 16:00 hours is stored in the appointment information of the survey. The scheduler will take this two-hour time difference into account, making sure that the form is delivered at the correct time.

When supervisors are viewing case information using BtMana or interviewers are viewing Case Summary information before running the data entry program, all times are automatically converted by Blaise into interviewer time.

Treatment types

The CATI system distinguishes a number of treatment types, which are applied to forms based on the dial result. Blaise recognises the treatments *no-answer, busy*, and *appointment.* Forms with the other dial results—*disconnected*, *other*, *response*, and *nonresponse*—are considered concluded, and therefore no treatment is needed. Treatments are assigned in the CATI Specification Program and in the data model.

Appendix A: Command Line Parameters

This appendix lists command line parameters in Blaise®. Command line parameters govern the way each system component works when it is run. These programs can be run from Maniplus, or from a Windows® batch process such as VBScript or Jscript using the Windows® Script Host capability, or from other commercially-available batch utilities such as WinBatch™.

Command line prepare utility (B4CPars.exe)

Command Line Parameter	Description
<file-name>	Set <file-name> as data model or Manipula/Maniplus setup or project file to prepare This program supports wild cards. For instance B4CPars *.bla will prepare all .bla files.
/A<+|->	Wait for key at end. Default /A-
/C<+|->	Check layout identifiers. Default /C-
/H<foldername[;...]>	Set meta search path
/M<modelibrary>	Set mode library name
/O<foldername>	Set output folder
/Q<+|->	Run parser in quiet mode. Default /Q-
/S<foldername[;...]>	Set include file search path
/T<option-file>	File with options
/U<metaname[,metaname]>	Set meta names for uses
/W<foldername>	Set working folder
/X<+|->	Sets optimized checking on/off. Default is /X+.
@<filename>	Prepares all files named in <filename>

Cameleon (cameleon.exe)

Command Line Parameter	Description
<file-name>	Set <file-name> as script to process.
/B	Suppress all dialogs and never wait for a user key press.
/D<file-name>	Set <file-name> as name of the data model to use.
/H<folder[;folder]>	Set meta search path.
/P<parameter[;parameter]>	Set parameters.
/T<folder>	Set <folder> as write-to folder.
/W<folder>	Set <folder> as working folder.

CATI Emulator (btemula.exe)

Command Line Parameter	Description
<file-name>	Set <file-name> as emulator option file to use.
/H<folder[;folder]>	Set meta search path.
/E<folder[;folder]>	Set external file search path.
/W<folder>	Set <folder> as working folder.

CATI Management Program (btmana.exe)

Command Line Parameter	Description
<file-name>	Set <file-name> as CATI specification to use.
/B	Create daybatch in batch mode.
/BN	Create a daybatch for the next defined survey day.
/H<folder[;folder]>	Set meta search path.
/P<password>	Use <password> to create daybatch (if applicable).
/W<folder>	Set <folder> as working folder.

CATI Specification Program (btspec.exe)

Command Line Parameter	Description
<file-name>	Set <file-name> as CATI specification to use.
/H<folder[;folder]>	Set meta search path.
/W<folder>	Set <folder> as working folder.

Control Centre (blaise.exe)

Command Line Parameter	Description
<file-name>	Open <file-name> in the Control Centre.

Data Entry Program (dep.exe)

In addition to the usual command line approach for passing parameters to an application, described below, Dep supports Blaise® command line option (`.bcf`) files, also called the @-option. All the command line options for a process are set in a text file, for example, myparms.bcf, and invoked with Dep.

```
Dep @myparms.bcf
```

Command line files are described in the next section of this appendix.

Command Line Parameter	Description
<file-name>	Set <file-name> as data model to use.
/B	Open in Browse Forms mode.
/C<file-name>	Read configuration file <file-name>.
/D	Disable CATI (if applicable).
/E<folder[;folder]>	Set external file search path.
/F<file-name>	Use <file-name> as main data file.

Command Line Parameter	Description
/G	Open in Get Form mode. When used with /K, reads information from /K. When using /G in combination with the command line option /Y<filter> (the form status filter option) the DEP will load automatically the first form that complies with the specified filter. It is possible to leave the filter empty. In this case the first available form in the database will be loaded. This functionality is available only if you don't specify the value of a key on the command line (so the option /K is not present). Example: DEP LFS00A /G /YDS will load the first available dirty or suspect form.
/H<folder[;folder]>	Set meta search path.
/J	Open with form with given join ID only. Used only in combination with /K.
/K<key-value>	Get the first form with the <key-value> for the primary key or internal key (see also /G, /J, and /N).
/L<number>	Set interview language as <ID number>. The default is 1.
/M<file-name>	Read the menu file <file-name>. The default is depmenu.bwm, or catimenu.bwm in case of CATI.
/N	Open and create a form with given key. Used only in combination with /K.
/O	Disable image link.
/P<number>	Use page layout <number> at start. The default is 1.
/R	Run in read-only mode, editing disabled.
/RE	Run in read-only mode, but allow edit to experiment with form.
/S<parallel-name>	Go directly to the specified parallel when entering the form. This option will only work when the key page is not activated when starting the DEP and when the specified parallel can be reached.
/T<number>	Use behaviour toggles <number> at start. The default is 1.
/W<folder>	Use <folder> as working folder.
/X	Exit DEP after editing the first form.
/Y<cdsn>	Form types filter. <cdsn> is any combination of the letters c=Clean, d=Dirty, s=Suspect, n=Not checked.
/Z	Go directly to last field on the route that needs to be answered. This option is similar to pressing the END key directly after loading a form.
/!	Activate the watch window.
@<filename>	Start with command line option file (see section A.1)

Hospital (hospital.exe)

Command Line Parameter	Description
<file-name>	Set <file-name> as script to process.
/C	Check the data file in batch mode.
/H<folder[;folder]>	Set meta search path.
/L<diagnose log-file>	Set diagnose log file
/M<file-name>	Set <file-name> as data model to use.
/R	Rebuild the data file in batch mode.
/W<folder>	Set <folder> as working folder.

Manipula/Maniplus (manipula.exe)

In addition to the usual command line approach for passing parameters to an application, described below, Manipula supports Blaise command line option (`.bcf`) files, also called the @-option. All the command line options for a process are set in a text file, for example, myparms.bcf, and invoked with Manipula.

```
Manipula @myparms.bcf
```

Command line files are described in the next section of this appendix.

Command Line Parameter	Description
<file-name>	Set <file-name> as setup to use.
/A	Wait for key at end.
/B	Never wait for a key press.
/C<file-name>	Read configuration file <file-name> instead of the default manipula.miw.
/D<file-name>	Set <file-name> as day file.
/E<folder>	Set the path to search for external data files used during checkrules. Path only applies to external files with no path specified. This command line option is similar to the /E command line option of the data entry program. The system will also search the folder specified by the /F parameter (the read-from folder) when needed.
/F<folder>	Set <folder> as read-from folder.
/H<folder[;folder]>	Set meta search path.
/I<file-name[,file-name]>	Set input and update file name(s).
/M<file-name>	Set <file-name> as parameter file.
/O<file-name[,file-name]>	Set output file name(s).
/P<parameter[;parameter]>	Set parameters.
/Q	Run in quiet mode.
/R<file-name>	Set <file-name> as message file.
/T<folder>	Set <folder> as write-to folder.
/W<folder>	Set <folder> as working folder.
/X<folder>	Set <folder> as temporary sort file folder.
/!	Activate the watch window.
@<filename>	Start with command line option file (see section A.1)

Blaise Command Line Option Files

To solve problems with very long command lines, the DEP and Manipula both support Blaise command line option (`.bcf`) files. This option is also called the @-option. The @-option requires a file name:

```
@<file-name>…
```

With the @-option you instruct the system to read the different command line options from the named command line option file <file-name>. This command line option file has an ini-file structure. All DEP and Manipula command line option are listed below.

Creating a Blaise command line option file

An easy way to create or change a Blaise command line option file is by making use of the Run parameters dialog in the Blaise Control Centre. This dialog can be used as an editor for a Blaise command line option file. With the load button you can read the values store in such a file and with the store button you can write such a file.

To edit the Data Entry options you need to focus the Data Entry tab. When this tab is focussed pressing the store button will prompt you for a file name and after providing that name all current Data Entry options will be stored in the named command line option file. Using the load button will read all Data Entry options from a named option file.

To edit the Manipula options you need to focus the Manipula tab. When this tab is focussed pressing the store button will prompt you for a file name and after providing that name all current Manipula options will be stored in the named command line option file. Using the load button will read all Manipula options from a named option file.

The current value of the meta search path in the Project options dialog will also be stored. Based on the setting of the working folder in the project options dialog, the name of the primary file (if applicable) or the file name corresponding with the currently focussed MDI window, the system will determine the value of the working folder. This value will also be stored.

Here is an example to illustrate its use. Suppose you want to execute the following command:

```
DEP c:\mymeta\example /fc:\mydata\example /mc:\misc\demo.bwm /x
```

This is equivalent to executing DEP @`example.bcf` where `example.bcf` looks as follows:

```
[DepCmd]
DataModel=c:\mymeta\example
DataFile=c:\mydata\example
MenuFile=c:\misc\demo.bwm
ExitDep=1
```

The @-option can be mixed with normal command line options. For instance:

```
DEP @example.bcf /mc:\misc\other.bwm
```

In this example the DEP will use the menu file mentioned on the command line.

Important: The command line options are evaluated from left to right. If an option is present on the command line and also in the option file the last value encountered will be used.

So:

```
DEP /mc:\misc\other.bwm @example.bcf
```

will use the menu file mentioned in `example.bcf` if present. If the menu file is not specified in `example.bcf` the name specified on the command line will be used.

If you do not specify an extension the default extension `.bcf` (Blaise Command line option File) will be used. So DEP @`example.bcf` is equivalent to DEP @example.

The @-option is also supported by the CALL and the EDIT instruction/method in Maniplus.

DEP syntax for Blaise Command Line Option File

Statement	Description	Command line Parameter
[DepCmd]	Declares that the following lines contain commands used for DEP.EXE	
DataModel=<file-name>	Set data model name to use	
BrowseMode=0 or 1	Open in Browse Forms mode	/B
ConfigFile=<file-name>	Read ConfigFile instead of the default dep.diw	/C
DisableCATI=0 or 1	Disable CATI (if applicable)	/D
ExternalSearchPath=<folder[;folder]>	Set external file search path	/E
DataFile=<file-name>	Use DataFileName as main data file name	/F
GetMode=0 or 1	Open in Get Form mode. If available reads information from Key=	/G
Language=<number>	Active language number	
MetaSearchPath=<folder[;folder]>	Set meta search path	/H
UseRecordNumber=0 or 1	Use Key = value as internal recordnumber and open with that form only	/J
Key=<key-value>	Get the first form with the key value for the primary key or internal recordnumber. See also UseRecordNumber, GetMode and CreateForm	/K
Language=<number>	Set interview language number. The default is 1	/L
MenuFile=<file-name>	Set the menu file name. The default is depmenu.bwm, or catimenu.bwm in case of CATI	/M
CreateForm=0 or 1	Create a form with given key and open it. Use only in combination with Key=	/N
LayoutSet=<number>	Use page layout LayoutSet at start. The default is 1	/P
DisableImageLink=0 or 1	Disable IMGLink (if applicable)	/O
ReadOnly=0 or 1	Run in read only mode	/R
Readonlyedit=0 or 1	Run in read only mode, but allow edit of form without save	/RE
StartParallel=<Parallel-name>	Set the parallel that needs to be activated	/S

Statement	Description	Command line Parameter
ToggleSet=<number>	Use behaviour toggles ToggleSet at start. The default is 1	/T
WorkFolder=<folder>	Set WorkFolder as working folder	/W
ExitDep=0 or 1	Exit DEP after editing the first from	/X
SelectStatus=	Set the records to select based on the select status: C=Clean, D=Dirty, S=Suspect, N=Notchecked	/Y
Watchwindow=0 or 1	Activate the watch window	/!
GotoEnd=0 or 1	Same behaviour as pressing the END key in a form	/Z

Manipula syntax for Blaise Command Line Option Files

Statement	Description	Command line Parameter
[ManipulaCmd]	Declares that the following lines contain commands used for MANIPULA.EXE	
Setup=<file-name>	Set the name of the setup to use	
WaitForKey=0 or 1	Wait for key at end	/A
BatchMode=0 or 1	Never wait for a key press	/B
ConfigFile=<file-name>	Read ConfigFile instead of the default Manipula.miw	/C
DayFile=<file-name>	Set day file name	/D
ExternalSearchPath=<folder>	Set external file search path	/E
InputFolder=<folder>	Set folder to read from	/F
MetaSearchPath=<folder[;folder]>	Set meta search path.	/H
InputFile=<file-name[;file-name]>	Set input and update filenames	/I
OutputFile=<file-name[;file-name]>	Set output filenames	/O
Parameter=<parameter[;parmeter]>	Set parameters	/P
RunQuiet=0 or 1	Run in quiet mode	/Q
MessageFile=<file-name>	Set message file name	/R
OutputFolder=<folder>	Set folder to write to	/T
WorkFolder=<folder>	Set WorkFolder as working folder	/W
Watchwindow=0 or 1	Activate the watch window	/!
SortWorkFolder=<folder>	Set folder for temporary sort files	/X

Appendix B: Files in Blaise

This appendix describes the different kinds of files that are used in Blaise® and provides some suggested folder structures. We use the National Commuter Survey as an example, where NCS is used for instrument files and `ncsdata` is used for the data files. The following types of files are discussed:

- Instrument files
- Blaise® data files
- External data files
- Configuration files
- Blaise® system files for stand-alone or remote operation
- Manipula files for stand-alone or remote operation
- Files for Maniplus stand-alone or remote operation
- Files for distribution for an application.
- Source code files
- Directory structures
- CATI files

Instrument Files

The following are the compiled instrument files.

Name	Description	Example
Meta information file	Holds meta-information, data definition, rules.	ncs.bmi
Data model file	Screen layout information.	ncs.bdm
Extended meta file	Parallel text definitions	ncs.bxi

Blaise Data Files

The following Blaise data files are always present.

Name	Description	Example
Main data file	Holds all Blaise data.	ncsdata.bdb
Join key file	Internal index file, generated by the Blaise system.	ncsdata.bjk
Information file	Internal file information file, generated by the Blaise system when data files are first created.	ncsdata.bfi

The following Blaise data files are not always present.

Name	Description	Example
Primary key file	Index file based on primary key.	ncsdata.bpk
Secondary key file	Index file based on secondary keys.	ncsdata.bsk
Remark file	Remark file.	ncsdata.brd
Remark index file	Indexes remarks.	ncsdata.bri
Trigram file	Trigram data file.	ncsdata.bdt
Trigram index file	Indexes trigrams.	ncsdata.bit

The remark files are created the first time a user makes a remark. The primary key and secondary key files are created if primary or secondary keys are declared in the data model. The trigram file and trigram index files are created if a secondary key of type trigram is declared in the data model.

The following file can be created by the user in the Database Browser. The user can select certain fields in a Blaise database to view, save that view to a file, and then open the view file in the Database Browser.

Name	Description	Example
Database view file	View of selected fields in the Database Browser.	ncsview.bdv

External Data Files

If the NCS instrument refers to externally held data, these are the external files that would be required by the application.

Name	Description	Example
External meta file	Holds external meta information.	ncsext.bmi
External main data file	Holds all external Blaise data.	ncsext.bdb
External primary key file	Index file based on primary key.	ncsext.bpk
External join key file	Internal key index file.	ncsext.bjk
External information file	Internal file information file.	ncsext.bfi
External secondary key file	Index file based on secondary keys, used for lookups.	ncsext.bsk
External trigram file	Data file for trigrams, used for trigram lookups.	ncsext.bdt
External trigram index file	Indexes trigrams, used for trigram lookups.	ncsext.bit

If the external file will not be used for data entry purposes by other applications, the external main data file (`.bdb`) can be flagged with the *ReadOnly* attribute. This will speed up the input/output process on the external file.

DEP Customisation Files

These are customisation files that can be used to control the look and behaviour of the DEP.

Name	Description	Example
Mode library	Holds behaviour toggles, screen layout styles, text, font, multimedia, and other behaviour settings.	modelib.bml ncslib.bml
Menu file	Determines the menu options, short-cut keys, and speed buttons available for the DEP user.	depmenu.bwm ncsmenu.bwm
DEP configuration file	Holds behaviour toggles, text and font enhancements, multimedia, and other behaviour settings. When applied, settings here override the settings in the mode library file under which the data model was prepared.	dep.diw ncs.diw

Note that these files can have their own names, specific to a project. It is not unusual to invoke one modelib file for data collection and another for data editing. The same can be said for the menu and configuration files. You can create different `.diw` files for different screen resolutions on different computers.

Data Entry Program Files for Stand-alone or Remote Operation

Name	File
Data Entry Program	dep.exe
Data Entry Program Help	dep.hlp

Manipula/Maniplus Files for Stand-alone or Remote Operation

Name	File
Manipula	manipula.exe

Application-specific compiled setup files compiled from Manipula/Maniplus programs are files with the `.msu` extension. These are used with `manipula.exe`. If there was a change to the meta information file that was used, be sure to re-prepare any `.msu` files.

For Maniplus setup files, the following extra files might be needed:

Name	File
Help for Data Entry Program	Dep.hlp is required only if you use EDIT in the setup.
Nested setups	<setup-file>.msu

Files for Distribution for an Application

If you are running an application on laptops or on a local area network, the laptops and network do not need all system files that come with the Blaise distribution. Files that are needed for distribution to a remote site include:

- The appropriate Data Entry Program, Manipula, and Maniplus files mentioned above. When you perform all data entry/interviewing through EDIT in Maniplus setups, you don't need to install the `dep.exe`.
- Instrument files mentioned above.
- External files mentioned above (some might not exist, such as the trigram files, if they are not part of the application).
- Data files if they are pre-loaded. Otherwise, the system will create them when the instrument is first invoked.
- The compiled menu file (such as `depmenu.bwm`) and the Data Entry Program configuration file `dep.diw`. There is no need to send the mode library file to a remote location because it affects the instrument during preparation only.

Source Code Files

Name	Description	Example
Master file	Starts with DATAMODEL and ends with ENDMODEL.	ncs.bla
Included files	Any file that is brought into the instrument with an INCLUDE statement.	secta.inc
Procedure files	Include file that holds a Blaise procedure.	m_of_n.prc
Library file	A file that holds type definitions. Starts with LIBRARY and ends with ENDLIBRARY. Needs to be prepared to a .bli file.	ncs.lib

Note that the extensions are suggestions only and are not syntactically required. It is preferable to keep extensions of kinds of files distinct so that they can be distinguished easily.

Folder Structures

For an application, keep a folder structure that helps organise all the files. For example:

Folder Structure	Use
Ncs\Source	Application-specific master and INCLUDE files.
\Inst	Compiled instrument files.
\Data	Production data files.
\Pracdata	Practice data files.
\Extern	External information.
\Manipula	Manipula and Maniplus files.
\Bat	Batch files.
Blib\Genblock	Organisation-wide general blocks.
\Typelib	Organisation-wide general types.
\Genproc	Organisation-wide Blaise procedures.

CATI Call Management System Files

The CATI Call Management System produces its own set of files:

Name	Description	Example
Daybatch	Holds all daybatch information.	ncsdata.btd
Daybatch index	Daybatch index.	ncsdata.bti
Specification file	All CATI survey specification parameters. Includes the dial menu, interviewer and group names, valid days of the survey, time zone definitions, and so on.	ncsdata.bts
History file	Keeps track of all call and dial attempts.	ncsdata.bth
Counts file	Keeps track of all counts.	ncsdata.btc
Log File	The system uses this log file to write messages about occurring events that might be relevant for the CATI management system, like the start of an interviewer session, the creation of a daybatch, the occurrence of error situations and the end of a session.	ncsdata.log
Daybatch ASCII	ASCII representation of the binary daybatch file above. This file is generated only if the *View daybatch*, *Browse* has been invoked in the CATI Management Program.	ncsdata.tdb

The `.bts` file, which holds all CATI definition parameters, can be used from one survey to the next. This saves the trouble of re-entering all the information that is used from survey to survey.

Index

A

ACTIVELANGUAGE
Key word · 145
Advanced Manipula · 399
AFTER
Location key word · 264
ALIEN
DLL key word · 237
Alien procedure · 236, 237, 238
Alien router · 237, 238, 287
Used with DLL · 236
Alphabetic search
As lookup · 227
Use of · 228
AND
Key word · 124
Answer attributes · 93
Block level · 96
Data model level · 96
Don't know · 93
Empty · 93
Refusal · 93
Appointments
Blocks in data model · 185, 190, 191
Disabling default appointment dialog · 529
Hard · 564, 569
Making in DEP · 485
Parameters in CATI specification file · 508
Preference · 564, 569, 580
Viewing in CATI Management Program · 539
Arithmetical expressions · 124
Examples of · 123
ARRAY
Key word · 88, 158
Array methods
DELETE · 183
EXCHANGE · 183
For blocks · 183
INSERT · 183
Arrays · 158
Of blocks · 195
Performance issues · 200
ASCII
Key word in Manipula · 381
ASCIIRELATIONAL
Key word in Manipula · 381
ASCIIRelational files · 367, 378, 437
ASK
Key word · 108
Assignments · *See Computations*
AT
Location key word · 264
ATTRIBUTES
Key word · 96
Audio-CASI · *See Multimedia*
Audit trail · 238, 247
Audit trail DLL · 239, 288
Contents of · 246
Date and time stamp · 241
Invoking · 288
Mode library file settings · 245
Processed by Manipula · 247
Recording user actions · 240
Requirements · 239
Summary of · 247
Turning on and off · 247
Uses of · 238
AUTOCOPY
Key word · 377, 450
AUTOREAD · 410
Key word · 377
Auxfields · 102, 399
Compared with locals and fields · 106
Examples of · 102
Mixing with FIELDS section · 103
Placement for good performance · 199
Used as label · 103
Uses of · 102
AUXFIELDS
Key word · 102, 379
Auxfields section · 102. *See also Auxfields*
Auxiliary fields · *See Auxfields*

B

B4CPars.exe · 42, 585

bdv file (saving a view in Database Browser) · 55, 596
BEFORE
  Location key word · 264
BLAISE
  Key word · 378
Blaise data files
  Compatibility · 147
    Causes of incompatibility · 148
    Methods to handle incompatibility · 149
  Exporting blocks of · 437
  Extending using Manipula · 385
  Importing blocks of · 428
  Initialising · 385, 499
  Reformatting with Manipula · 420
  Updating data definition with Manipula · 150
  Viewing · 50, 55, 596
Blaise language · *See also The Reference Manual*
  Key words · 61
  Overview of · 59
BLAISEUSER
  Environment variable · 417, 517
BLOCK
  Key word · 155
BLOCKEND
  Location key word · 264
Blocks · 67, 153
  Administrative · 185, 190, 197
  Appointment · 185, 190
  Array methods for · 183
  Arrayed · 158, 195, 197
  As field type · 67, 90, 157
  As included files · 173
  Block computations · 182, 412
  Block history · 409
  Child · 165, 166
  Dot notation in · 157, 161
  Embedded · 439
  Enumerated instances of · 159
  For testing · 153, 167
  Making EMPTY with block computation · 184
  Management · 185
  Manipula and · 154
  Menus as alternative to parallel blocks · 193
  Multiple separate · 162, 163
  Naming · 157, 162
    Block field name · 156
    Block type name · 156
  Nested · 164, 166, 195
  Nonresponse · 185, 188
  Parallel · *See Parallel blocks*
  Parameters in · 167, 169, 193
  Parent · 165
  Passing information to · 161, 162
  Protecting from change · 181
  Selective checking and · 198
  Syntax · 155
  Text at block level · 160, 161, 199
  Uses of · 153
BLOCKSTART
  Location key word · 264

C

Cameleon · 9, 453, 458
  Blocks as basis for · 154
  Command line parameters · 586
  Examples · 468
  Manipula and · 454
  Meta information files and · 453
  Output samples · 460
    Param.cif · 471
    SAS.CIF · 463
    SPSSPC.CIF · 460
  Programming · 465
    Basic concepts · 466
    Meta data loops · 473
  Run parameters · 459
  Running · 457
  Translators supplied with Blaise · 456
  Use of text as field description · 71
CARDINAL function · 81
CASE OF
  Structure in Manipula · 384
CATI Call Management System · 477. *See also CATI Management Program; CATI Specification Program*
  Blaise CATI concepts · 478
  Call scheduler · 565
    Activating forms · 571
    Assigning priorities · 481, 568, 582
    Assigning statuses · 481
    Future priority · 581
    Routing back forms · 567, 571
  CATI Management Program · *See CATI Management Program*
  CATI specification program · *See CATI Specification Program*
  CATI terminology · 580
  Technical details · 563
  Time slices · 491, 583
  Treatment types · 573
    Appointment · 575

Busy · 574
Exceptions to general rules · 575
No answer · 573, 584
CATI data models · 490, 551
Blocks in · 490, 498
Appointment · 497, 498, 553
TAppMana · 491
TCallMana · 491, 510
TSliceMana · 491
CAPI compatibility · 557, 558
CATI menu · 344, 489. *See also Menu file*
Disabling CATI mode · 349, 558
Fields in · 493
Quota · 496
Telephone · 553
Files needed for · 576
INHERIT CATI · 490, 553
Initialising · 499, 553
Quotas · 483
Running in the DEP · 483, 556
Appointments · 485, 487, 488
Dial screen · 484
Testing · 559
CATI Emulator · 419, 559
Command line parameters · 586
Manipula to produce a script · 247
CATI Management Program · 533, 555
Command line parameters · 586
Daybatch · 559
Browsing · 541
Creating · 535
Rules for inclusion in · 563
Scheduling · 541
Setting parameters · 507, 530, 533
Viewing · 536, 537, 542
Forms · 543, 566, 567
Activating · 541
Selecting for further treatment · 544
History file · 547, 577, 581
Log file · 550
Running outside the Control Centre · 550
Setting environment options · 546
Summary of calls · 542
Viewing active interviewers · 546
Viewing appointments · 539
CATI Specification Program · 500, 555
CATI specification file · 582
Command line parameters · 587
Defining · 501
Crews · 505
Daybatch select · 530
Daybatch sort · 533
Dial menu · 511
Field selection · 513, 517
Interviewers and groups · 520
Parallel blocks · 528
Quotas · 525, 528
Survey days · 503
Time slices · 523
Time zones · 522
CHECK
Key word · 60, 120
Checks · 120, 121
Child block · *See Blocks, child*
Classification type · 205, 206
Building · 208
Classify methods · 210
Descriptive text in · 207
Dynamic coding frame · 209
Level names · 209
Use of Manipula to build · 208
CLASSIFY
Key word · 210
Classify methods · 206, 210
CLASSTOSTR
Key word · 213
Coding
Coding dialog · 211
Controlling size and location · 212
Using · 210, 212
From an open question · 212
Hierarchical coding · 205
Accessing external data based on code · 213
Classification · *See Classification type*
Classification fields · 88
Classify method · 210
Converting code to string · 213
Used in combination with lookups · 232
Types of · 205
Using lookups · 230
Command line parameters · 585
Command line prepare utility · 42, 585
Computations · 132
Compute instruction · 67
Dates and · 86
Definition of · 132
Enumerated fields and · 78
Fields as expressions · 132
Location in rules section · 132
String fields and · 74
Compute instruction · *See Computations*
Concurrent interviewing
Through parallel blocks · 191
Conditional rules · 113
Defining · 113

Edit checks in · 114
Involving fields · 120
ELSE instruction in · 114
ELSE-IF instruction in · 115
Error text in · 119
Fields listed in · 117
IF condition in · 113
Route instructions in · 113
Specifying other choice · 116
CONNECT
Key word · 395, 446, 450
Control Centre · 7, 18
Command line parameters · 587
Configuring the Tools menu · 32, 34
Data file management · 31
Database Browser · *See Database Browser*
File types in · 9
Opening files · 9
Setting environment options · 15, 47, 56
Structure Browser · *See Structure Browser*
Text editor · 11
Opening files · 10
Shortcut keys · 12
Viewing file history list · 10

D

Data Entry Program · 267
Behaviour modes · 274, 275, 276
Checking · 275
Defining · 276
Error reporting · 275
Mode library file settings · 282, 292. *See also Mode library file*
Routing · 274
CATI data models and · *See CATI data models, Running in the DEP*
Command line parameters · 587
Customising · 268, 277, 281, 331, 346. *See also DEP configuration file; Mode library file; Menu file; Data Entry Program, Screen design*
FieldPane · 271
FormPane · 269
Grid · 270
InfoPane · 272
Page · See FormPane
Run parameters · 347
Running outside the Control Centre · 364
Screen design · 267, 277. *See also Mode library file; Menu file*
Common screen layout tasks · 317, 319, 322
Examples · 259, 317
Layout section in data model · 259, 346. *See also Layout section*
Moving horizontal dividing line · 346
Screen layout factors · 345, 346
Screen resolution · 100, 346
Speedbar · 273, 336, 343
Using · 350
Browse forms · 357
Don't know · 355
Entering responses · 353
Error viewing · 360, 361, 362
Get form · 357
Invoking a behaviour mode · 351, 352
Multimedia · 363
Navigating between forms · 357
Refusal · 355
Remarks · 355
Shortcut keys · 354
Start asker · 357
Sub forms · 356. *See also Parallel blocks*
Switching languages · 362
Window components · 268, 273
Data export · 437
Blocks as unit for · 154
Manipula and · 367. *See also Manipula setups*
Data import · 428, 430, 434
Data models · 9, 59
Auxfields section · 102
Creating organisation in · 156, 199
Describing an external file · 216
Examples and supporting files · 151, 202
External · *See External files*
Extracting a mode library file · 324
Fields section · 70. *See also Fields; Fields section*
Hierarchical · 67, 195
Blocks as basis · 153
Included files · 9. *See also Included files*
Layout section · 259. *See also Layout section*
Locals section · 104
Mini-data models · 153, 182
Placement of edit checks in · 125
Preparing · 18, 324
Properties
Specifying text for parallel blocks · 193, 327
Rules section · 107. *See also Rules*
Running · 21
Settings section · 139. *See also Settings section*

Tab stops in · 100
Text enhancements in · 284. *See also Text enhancements*
Type section · 74, 88. *See also Types*
Data recovery
Hospital utility · 40
Using audit trail · 239, 247
Data storage
Blocks as unit for · 154
Database Browser · 50, 297
Detail panel · 51
Opening · 51
Record filter · 54
Saving a view · 55, 596
Searching on keys · 52
Selecting fields to view · 54
Setting options · 56
DATAMODEL
Key word · 61
Date formats in Blaise · 85
DATETYPE
Key word · 85
DELETE
Array method · 183
DEP configuration file · 277, 331, 333
Applying · 277, 335
Editing · 332, 335
Mode library file and · 280, 332, 345
DEP Configuration Program · 331, 332
Opening a configuration file · 333
Opening a data model · 333
DITTO
Key word · 201
Ditto function · 201, 294
DK
Key word · *See DONTKNOW*
DLLs
Alien procedure · 237
Executing DLL procedure · 238
Alien router · 237, 238
Audit trail DLL · 239
Procedures and · 147
Referencing · 237
Requirements for Blaise · 237
Uses of · 236
DO · *See FOR loop*
DONTKNOW · 93, 95
In IF condition · 94
Stored as status · 94
Dot notation · 101, 107, 147, 157
For nested blocks · 165
In IF condition · 166
To reference field or auxfield in a block · 161
Used with type libraries · 91
DUMMY
In tables · 178
Key word · 137
Key word in layout section · 264
Multiple in a row · 137
DYNAMIC
Classification type key word · 209
dynamic type · 83, 93

E

Edit checks · 60, 63, 120
Adding or deleting edits · 128, 130
Arithmetical expression · 123
Between different blocks · 163, 200
Checks · 121
Defining · 121, 122
ERROR function · 123
In conditional rules · 114
INVOLVING function in · 127
Manipula and · 369
Placement in data model · 125
Rules for use of · 127, 128
Rules of precedence of operators · 125
Signals · 121
Simple edits · 124
String expressions · 124
Toggling edit severity · 131
Use of procedure to implement · 233
Variable text fills in · 126
Edit Jump Dialog · 72
ELSE
Key word · 114
ELSEIF
Key word · 115
EMBEDDED
Key word · 439, 440
EMPTY · 75
Answer attribute key word · 93
In IF condition · 94
Suggested use of · 94
ENDBLOCK
Keyword · 155
ENDDO · *See FOR loop*
ENDIF
Key word · 113
ENDMODEL
Key word · 61
ENDPROCEDURE
Key word · 233

ENDTABLE
Key word · 176
Environment settings in Windows registry · 417
ENVVAR
Environment variable function · 418
ERROR
Alternative edit check declaration · 123
Multimedia key word · 249
Error text
In CHECK and SIGNAL · 119
In IF conditions · 119
Variable text · 126
ERRORCOUNT
Key word · 408
Errors · *See also Data Entry Program; Edit checks*
Default · 120
Hard · 120
Soft · 120
Example files · 5
Examples
Cameleon · 466, 468
Parameters · 468
WesVar.CIF · 468
CATI data model · 551
Data models · 9, 151, 202, 454
Help files · 258
Manipula · 372, 384, 398, 424, 450
EXCHANGE
Array method · 183
EXPORT
Key word · 170
Export parameters · *See Parameters*
External files · 214, 219, 224
Converting to Blaise format · 215
Data file as · 214, 217
Data model as · 214, 215, 217
Data model to describe · 216
Externals section · 215, 217, 218
File methods · 220
OPEN · 223
READ · 215, 222
RESULT · 223
SEARCH · 215, 220
Memory considerations · 224
Multiple files · 224
Placement for good performance · 199
Requirements · 213, 214
Restricting external fields · 219
Searching for a record · 220
Switching between files with OPEN · 223
Uses of · 214, 223
Uses section · 215, 217, 218
EXTERNALS
Key word · 215
Externals section · 215. *See also External files*

F

Field tag · 73. *See also Fields*
FieldPane · 271
Defining screen layout possibilities · 262
Grid and · 304
Mode library file settings · 278, 306
FIELDPANE
Layout style key word · 264
Fields · 61, 62, 69
Answer attributes · 93
Arrayed · 88
As expressions · 132
As topic identifier in WinHelp · 255
Classification · 88
Compared with locals and auxfields · 106
Date · 85
DeltaDate notation · 86
Decimal · 77
Enumerated · 62, 78, 79
Making assignments · 78
Type compatability · 79
Values of · 81
Field definition · 59
Field descriptions · 71
Field tag · 73
Field text · 71, 72
New line in · 98
Integer · 76
Listing together · 73
Naming · 70, 71
Numeric · 62, 77
Open · 75, 76
Preventing return to field · 109
Real · *See Fields, Decimal*
Route field methods · 107
Set · 80
CARDINAL function in · 81
Direct assignment · 83
Elements used in a fill · 83
IN notation used with · 81
In type section · 90
Referring to a specific element · 81, 83
Testing for an item · 81
Type compatibility · 82
Values stored · 81
Statuses · 94, 133

String · 62, 74, 124
As open type · 76
Functions · 75
Maximum size of · 74
Text · *See Text enhancements*
Text fills in · 101
Time · 87
Types of · 74. *See also individual listings in Fields*
FIELDS
Keyword · 62
Fields section · 62, 70. *See also Fields*
Included files in · 175
File methods
For external files · 220
Manipula and · 403
Files in Blaise · 595
Blaise data files · 596
CATI Call Management System · 601
DEP customisation files · 277, 597
DEP files for stand-alone or remote operation · 598
· 146, 193, 328, 329, 331, 595
External data files · 597
File name extensions · 175
Folder structures · 600
For distribution for an application · 598
Instrument files · 595
Manipula/Maniplus for stand-alone or remote operation · 598
Meta information files · 9, 18
Cameleon and · 453
Data model · 18, 453, 501
Libraries · 18
Manipula · 18
Mode library file and · 279
Prepared files · 18
Project files · 27
Source code files · 18, 599
Filters
In Database Browser · 54
In Manipula · 448
FIXED
Key word · 379
Key word in Manipula · 381
Fonts · *See Text enhancements; Data Entry Program; Mode library file*
FOR loop · 134
In Manipula · 383
Local as control variable · 104
To define edit over instances of block · 163
FormPane · 269
Data density in · 154, 345
Labelling using auxfields · 103
Mode library file settings · 278
FORMSTATUS
Manipula key word · 408
FROM TO
Location key word · 264
Functions · 146
Examples of · 146
Handling of generated errors · 147
String fields · 75

G

Good programming practices · 93, 105, 116, 150, 201
Graphics · *See Multimedia*
Grid · 270, 300
Defining screen layout possibilities · 262
FieldPane and · 304
Mode library file settings · 278, 300, 305
GRID
Layout style key word · 264

H

HALT
Key word · 414
Hard error · *See Checks*
Help
Blaise help · 57
Context-sensitive in data model · 11
Mode library file settings · 288, 295
Question-by-question · 143, 254
Blaise language for · 258
Using WinHelp · 255, 256
Topic identifiers used with WinHelp · 255
Hierarchical coding · *See Coding*
Hierarchical data models · *See Data models*
Hospital utility · 40, 447
Command line parameters · 589

I

IF condition · 70, 113
Error text in · 119
Example of · 63
In Manipula · 383
IMAGE

Multimedia key word · 248, 249
IMPORT
Key word · 170
Import parameters · *See Parameters*
IN
Used with enumerated fields · 78
Used with set fields · 81
INCLUDE · 90
Key word · 153, 173
Included files · 173
Advantages of · 173
Include statement
At section level · 174
For multiple languages · 175
Format of · 173
Incorporating blocks into data model · 153
Multiple in one file · 174
Nested in included files · 174
InfoPane · 272
Defining screen layout possibilities · 262
Mode library file settings · 278, 309
INFOPANE
Layout style key word · 264
INPUTFILE
Key word in Manipula · 381
INSERT
Array method · 183
INTEGER
Key word · 77
Internal parameters · *See Parameters*
INVOLVING
Key word · 120, 127

J

Join ID · 348, 588

K

KEEP · 108
Applied to block level · 181
Protecting blocks and tables from change · 181
Used for time stamp · 88, 109
KEEPALL
Key word · 403
Keys · *See Primary key; Secondary key*

L

Label in FormPane · 103
Languages · 142
Defining in data model · 142, 144, 145
Help
As topic identifier for WinHelp · 255
Blaise help language · 257
Question-by-question help · 143
In question text · 72
Multimedia · *See Multimedia*
Role of INCLUDE · 175
Switching between · 252
Defining in data model · 144
In Data Entry Program · 145
Testing for current language · 145
TLanguage · 144
LAYOUT
Key word in data model · 263
Key word in Tables · 178
Layout section · 259
Implementing · 263
In data model · 259
Layout style key words · 264
FIELDPANE · 264
GRID · 264
INFOPANE · 264
Location key words and · 265
Layout styles · 262
Default · 264
Naming · 263
Types of · 262
Location key words · 264
AFTER · 264
AT · 264
BEFORE · 264
BLOCKEND · 264
BLOCKSTART · 264
FROM TO · 264
Layout style key words and · 265
Mode library file and · 262, 263, 301
Layout style key words · *See Layout section*
LIBRARY
Key word · 91
Linking files · 401
Locals · 104
Compared with fields and auxfields · 106
Control variable in a loop · 134
Placement for good performance · 199
Scope of · 105
Uses of · 104, 200
LOCALS · 69

Key word · 104
Location key words · *See Layout section*
Lookups · 224
Alphabetic search · 227
External lookup file · 225
For coding · 230
Hierarchical coding and · 232
Keys in the external lookup file · 226
Starting value for · 231
Switching between keys · 228
Trigram search · 227
Use of combined keys · 228
Uses of · 225
Verifying coding entries · 230
Looping
Local as control variable · 134
Through edit checks · 136
Through rules · 134

M

Maniplus · 3
Manipula · 9, 154, 367
Basic operation of · 377
Cameleon and · 454
Command line parameters · 416, 587, 590
Creating a setup · 369
Default settings · 377
Environment variables · 417
Example file structures · 404, 405, 407
File formats supported · 378
Filters in · 448
Improving performance · 445, 446, 448, 449, 450
LAN issues · 418
ACCESS · 418
BTEmula · 419
Concurrent tasks · 420
ONLOCK · 419
Manipula Wizard · 35, 370
Run parameters · 373
Running as a separate program · 397
Stopping · 414
HALT · 414
PAUSE · 415
READY · 414
Use to build classification type · 208
Uses of · 368, 444
Classification · 445
Exporting a data file · 386
Extending a data file · 385
Initialising a data file · 385
Test data set · 444
Manipula setups
AUTOCOPY=NO · 450
AUTOREAD · 410
Auxfields section · 388
Block computations · 412, 449
Block history · 409
CONNECT=NO · 450
Counting forms · 409
Date and time stamps · 391
Debugging · 415
Environment variables
ENVVAR · 418
USERNAME · 417
Exits from loops · 413
Exporting data · 437
ASCIIRelational files · 437, 444
Blocks of data · 442
Embedded and ordinary blocks · 439
File methods
KEEP · 403
KEEPALL · 403
WRITE · 403
WRITEALL · 403
Form correctness status · 408
ERRORCOUNT · 408
SELECTSTATUS · 408
SUPPRESSCOUNT · 408
Functions in · 413
Importing data · 428
Data in one file · 428
Data in separate files · 430
Two ASCII files at the same time · 434
Two ASCII files in two stages · 430
Input file section · 381
Linking files · 401
Blaise files as link files · 402
Dynamic link · 401
Static link · 401
Manipulate section · 377, 382, 407
CASE OF statements · 384
Control structures · 383
Expressions · 383
Functions · 383
Multiple manipulate sections · 384
Message file · 402
Output file section · 382
Preparing · 372
Print section · 390
Procedures in · 411
DLLs · 412
Manipula procedures · 411

Prologue section · 399
Reformatting files · 420
Many records to one · 423
One record to many · 420
Running · 373
Sections in · 379
Settings section · 391
File related · 394
Global · 391
SORT section · 389
TEMPORARYFILE section · 400, 449
UPDATEFILE section · 400
Uses section · 378, 379
MANIPULATE
Key word in Manipula · 383
Menu file · 277, 336
Applying · 277, 345
Editing · 337, 338, 343
Menu items · 338
Speed buttons · 343
User-defined menu items · 339, 340, 341, 343
System default · 277, 337
Menu Manager · 336, 337. *See also Menu file*
Mode Library Editor · 278, 279, 281. *See also Mode library file*
Extracting a mode library file · 324
Opening a data model · 282
Preparing a data model in · 283, 314
Printing mode library settings · 284
Saving a data model · 283
Viewing page and question properties · 315
Viewing pages in · 313, 314
Mode library file · 277, 278, 281
~dm file and · 279
Applying · 277, 323
Audit trail settings · 245, 288
Behaviour identifiers · 300, 334
Behaviour toggles · 282, 292, 293, 296, 297
Adding a toggle set · 299
Applying a toggle set · 300
Deleting a toggle set · 300
Colour settings · 284, 291
DEP configuration file and · 280
Extracting from a data model · 324
FieldPane settings · 300, 306, 307, 313
Font settings · 284
Default · 284
User-defined · 285
Grid settings · 300, 304, 305, 313
Grid for a table · 305
Help settings · 256, 288, 295
InfoPane settings · 300, 309, 313
Dialog boxes · 311, 312
Layout identifiers · 300, 302, 303, 334
Grid, InfoPane, FieldPane identifiers · 313
LAYOUT section and · 301
Layout sets · 300
Adding a layout set · 302
Applying a layout set · 303
Deleting a layout set · 303
Layout settings · 282, 300. *See also FieldPane settings; InfoPane settings; Grid settings*
Multimedia settings · 251, 298
Options settings · 284, 287
Printing · 284
Style settings · 284, 285, 287, 291
System default · 277
Monitor utility · 38
Multi-level rostering
Example instrument · 190
Multimedia · 248, 252
Audio fills · 253
Audio-CASI · 248
Declaring multimedia language · 249
Graphics · 248
Implementation · 248
Key words in · 249, 250
Mode library file settings · 251, 298
Uses of · 248
Video · 248

N

NEWCOLUMN
Key word · 137, 264
NEWLINE
Key word · 137, 264
NEWPAGE
Key word · 137, 178, 264
NODK · 93
NODONTKNOW
Answer attribute key word · 93
NOEMPTY
Answer attribute key word · 93
Nonresponse
As parallel block · 191
Blocks in data model · 185, 188
NOREFUSAL
Answer attribute key word · 93
NORF · *See NOREFUSAL*
NOT
Key word · 124

O

OPEN
  Field type key word · 75
  File method key word · 220
Operators · 123, 125
OR
  Key word · 124
ORD
  Function in Blaise · 79, 185
OUTPUTFILE
  Key word in Manipula · 382

P

Page-based presentation · 259. *See Layout section*
PARALLEL
  Key word · 191
Parallel blocks · 153, 191
  Accessing in the DEP · 192, 193, 288
  Assigning parallel status · 191
  CATI data models and · 497, 498
  For arrayed blocks · 192
  Parameters and · 193
  Specifying text for · 193, 327
  Types of · 191
  Unrouted · 192
Parameters · 169, 375
  Advantages of · 167
  Blocks and · 162, 167
  Definition of · 169
  Example of · 167
  Export · 170
  Forward checking · 200
  Import · 170, 199
  In administrative blocks · 190
  Internal · 172, 198
  List in rules section · 169, 171
  Mini-data models and · 182
  Names · 169
  Performance issues · 171, 172
  Procedures and · 233
  Selective checking and · 198
  Transit · 170
  Types of · 170
  Use in hierarchical data model · 195
  Uses for · 171
  Viewing in data model · 200
PAUSE
  Key word · 414
Performance issues
  Improving performance · 200
  Parameters and · 171, 172, 198
  Selective checking mechanism · 198
Pre-coded fields · *See Fields, Enumerated*
Prepare command · 18
  Command line prepare utility · 42, 585
PRIMARY
  Key word · 140
Primary file · 26, 173
Primary key · 139
  Basis of search in external files · 220
  Defining in data model · 140
  External file and · 213
  Searching on in Database Browser · 52
  Used in lookups · 225
  With secondary keys in lookups · 225
PRINT
  Key word · 379
PROCEDURE
  Key word · 233
Procedures · 233
  Example of · 234
  Manipula · 411
  To add edits to data model · 128
  Uses of · 233
Programmer's comments · 69
Projects · 25
  Creating · 25
  Opening · 27
  Options · 28
  Primary files · 26
  Project Manager · 25

Q

Question-by-question help · *See Help*
QUOTAREACHED
  Key word · 525

R

RANDOM function · 235
READ
  Key word · 213, 215, 220
READY
  Key word · 414
REAL
  Field type key word · 77
Real fields · *See Fields*

Reformatting files · 420
REFUSAL · 95
  Answer attribute key word · 93
  In IF condition · 94
  Stored as status · 94
REPEAT-loop
  In Manipula · 383
RESERVECHECK
  Key word · 128, 130, 233
Reserved words · 61
RESULT
  Key word · 220
RF · *See REFUSAL*
Rostering
  Multi-level · 195
  Tables used for · 154, 175
Route field methods · 107, 108
  ASK · 107
  KEEP · 107, 108, 109, 110, 112, 132, 181
  SHOW · 107, 181
Route instructions · 60, 62, 107
  Fields named in exclusive branches · 118
  In conditional rules · 113
Rules · 107
  Computations · *See Computations*
  Conditional · *See Conditional rules*
  Edit checks · *See Edit checks*
  Empty rules section · 138
  Execution of · 111
  Included files in · 175
  Layout elements · 60, 137, 138
    DUMMY · 137
    NEWCOLUMN · 137
    NEWLINE · 137
    NEWPAGE · 137
  Looping through · 134
  Manipulating arrays in · 183
  Omission of in a block · 138
  Precedence for operators in · 125
  Route instructions · *See Route instructions*
  Types of · 60, 62, 107
RULES · 107
  Key word · 62
Run command · 21
  Invoking · 21, 47
  Setting parameters · 22

S

Screen design · *See Data Entry Program, Screen design*
SEARCH
  Conditional use of · 221
  Key word · 213, 215, 220
SECONDARY
  Key word · 140
Secondary keys · 140, 446
  Defining in data model · 140
  Searching on in Database Browser · 52
  Trigram searching · 141
  Used in lookups · 225
  With primary key in lookups · 225
Selective checking · 172, 198
SELECTSTATUS
  Key word · 408
SET
  Key word · 80
SETLANGUAGE
  Key word · 144
Setting editor colours for Syntax Highlighting · 17
SETTINGS
  Key word · 96, 379
Settings section · 139
  Block level · 96
  Key fields in
    Primary key · 139
    Secondary key · 140
  Languages in · 142
  Placement in data model · 139
Shortcut keys
  Blaise text editor · 12
  Defining in menu file · 336, 339
  DEP default · 354
SHOW · 108
  Applied to block level · 181
SIGNAL
  Key word · 60, 120
Signals · 121
Soft error · *See SIGNAL*
SORT
  Key word · 379, 389
SOUND
  Multimedia key word · 248, 249
Speedbar · *See Data Entry Program*
STARTDATE
  Date function · 86
STARTTIME
  Time function · 87
STRING
  Key word · 74
String expressions · 124
Structure Browser · 43
  Detail panel · 45, 46

- Icons in · 44
- Opening · 43
- Setting options · 47

Styles · *See Layout section, Layout styles*
Subblock · *See Blocks, cild*
Subroutine
- DLL · 236

SUPPRESSCOUNT
- Key word · 408

Symbols
- ' (single quote) · 74, 75
- " (double quotes) · 71, 78, 100, 173, 208
- ( ) (parentheses) · 73, 78, 132, 389
- (colon) · 87, 169
- , (comma) · 78, 94, 140, 169, 208, 517
- . (dot notation) · 91, 157, 165
- / (slash) · 72, 208
- @ (at sign) · 99, 100
- @/ (at sign and slash) · 98
- @| (at sign and piping symbol) · 100
- [ ] (square brackets) · 81, 83
  - In Cameleon · 466
- [* (bracket and asterisk)
  - In Cameleon · 466
- ^ (caret) · 76, 100, 101
- _ (underscore) · 209
- { } (braces) · 69, 150
- | (piping symbol) · 230

SYSDATE
- Date function · 86

SYSTIME
- Time function · 87

T

Tab stops · 285, 286
- In data model · 100

TABLE
- Key word · 176

Tables · 175
- As special block · 176
- Columns in · 177
- Definition of · 176
- DUMMY in · 178
- Extremely large · 177
- Holes in · *See Tables, DUMMY in*
- Layout section · 178, 200
- Page breaks in · 178
- Pagination in · 178
- Protecting from change · 181
- Rows in · 176, 177
- Uses of · 154

TEMPORARYFILE
- Key word · 379, 400

Testing
- CATI Emulation · 559
- Creating a test data set · 444
- Role of mini-data models · 182
- Use of blocks in · 167

Text enhancements · 97. *See also Mode library file, Font settings*
- Font settings in mode library file · 284, 285
- Hard spaces · 98
- Marking in data model · 99
- New line command · 98
- Proportional vs. non-proportional font · 99
- Spaces in · 97
- Specifying in field text · 99

Time formats in Blaise · 85
Time slices · 491. *See also CATI data models; CATI Specification Program*
Time stamp · 87, 109
TIMETYPE
- Key word · 87

TLanguage · 144
TO · *See FOR loop*
Topic identifier
- Used in WinHelp · 255

TRANSIT
- Key word · 170

Transit parameter · *See Parameters*
TRIGRAM
- Key word · 141

Trigram search · 227, 228
- Mode library file settings · 297

TYPE · 88
- Key word · 69

Types · 88
- Advantages of · 89
- Blocks as · 157
- Defining · 93
- Examples of · 69
- Type libraries · 90, 91
  - Advantages of · 90, 91
  - Duplicate type names · 91
  - Field types allowed in · 91
- User defined · 88

U

UPDATEFILE
- Key word · 379, 400

USERNAME
    Environment variable function · 417
USES
    Key word · 215, 379
    Key word in Manipula · 378
Uses section
    For external files · 215
    For lookups · 229
    In Manipula · 378

V

Video · *See Multimedia*
VIDEO
    Multimedia key word · 249

W

WHILE-loop
    In Manipula · 383
Windows settings · 85, 347
WinHelp · 143, 255, 258, 295
    Topic identifiers · 255
WRITE
    Key word · 383, 403
WRITEALL
    Key word · 403

Y

Year 2000 compliance · 86