Xbase++ for Windows 95 Windows NT

Basic Users Guide

Alaska Software, Inc.
743 Horizon Court, Suite 107
Grand Junction CO 81506
USA

Alaska Software GmbH
Hauptstr. 71-79
65760 Eschborn/Frankfurt
Germany

January 1998
Printed in Germany

Contents

1. Installation

This chapter describes the installation process of your Xbase^{++} development package. It also provides you with important information about the system environment and configuration.

1.1. Removing previous Xbase^{++} installations

If you have a version of Xbase^{++} already installed, you should uninstall this software via the software control panel:

Click the Start button

Select Settings -> Control panel

Click the icon Add/remove programs

Select Xbase^{++} from the list in the Install/Uninstall tab page

Note that the uninstall program can only remove installed files. It does not remove files created after installation, such as OBJ or EXE files in the ..\SAMPLES directory, for example. If such files exist, a warning message is displayed stating that the program could not be removed. In this particular case, you can ignore the message.

When a previous version is removed, you should check the PATH, LIB and INCLUDE entries in the *AUTOEXEC.BAT* (Windows95) or in the registry key *HKEY_CURRENT_USER\Environment* (WindowsNT), respectively, for old values.

After a new installation, you have to recompile all the programs which you have compiled with a previous Xbase^{++} version.

1.2. Starting the installation program

Insert the CD in the CD-ROM drive and change to the directory \ENG to install the international version of Xbase^{++}. The program SETUP.EXE is found in this directory. It installs Xbase^{++} and guides you through the installation procedure prompting you for *Typical* or *Custom* installation.

Note: The Xbase^{++} installation does not change or replace any system DLL files. All DLL files required by Xbase^{++} are stored in newly created directories.

1.3. Typical installation

The Xbase^{++} development package is installed by default on drive C: with \ALASKA\XPPW32\ as the root directory <ROOT>:

```
<ROOT>
   +--BIN              Compiler and service programs
   +--BOOK             Online help
   +--LIB              Runtime library
   +--INCLUDE          #include files
   +--RESOURCE         Resource files
   +--RUNTIME          Runtime libraries
   +--SOURCE           Source code
        +--SYS         Source for System level files
        +--COMPAT      Compatibility functions for Clipper '87
        +--SAMPLES     Example programs
```

During installation your system configuration is changed to reflect the new directories. For Windows NT, the changes are applied to the registry database under *\HKEY_CURRENT_USER\Environment*, while *AUTOEXEC.BAT* is changed for Windows 95. The following registry keys or environment variables, respectively, are modified or created if missing:

PATH =	C:\ALASKA\XPPW32\BIN;C:\ALASKA\XPPW32\LIB;
LIB =	C:\ALASKA\XPPW32\LIB;
INCLUDE =	C:\ALASKA\XPPW32\INCLUDE;
XPPRESOURCE =	C:\ALASKA\XPPW32\RESOURCE;

Default file associations are defined for PRG, CH, XPJ and XFF files during a typical installation. Use custom installation to suppress this automatic feature.

1.4. Custom installation

With custom installation, you can select individual components of the Xbase^{++} package to be installed. Also, the definition of file associations can be selected individually.

CH	Include files are associated with Window's Notepad and the Xbase^{++} compiler. The compiler performs a syntax check for a CH file when invoked.
PRG	Source code files are associated with Window's Notepad and the Xbase^{++} compiler. A PRG file is compiled.
XFF	Form definition files are associated with the Xbase^{++} FormDesigner.
XPJ	Xbase^{++} Project files are associated with the ProjectBuilder. A left double click on an XPJ file builds a project.

Note: The file XPP.REG is available in the root directory of the Xbase^{++} installation. If file associations are not defined during installation they can be defined later by importing XPP.REG to the Windows registry using REGEDIT.EXE. Default file associations can also be changed by editing the XPP.REG file. For example, if you want source code files to be associated with the editor of your choice, replace the string "notepad.exe" in XPP.REG with the name of your editor and import the changed file to the registry.

Be aware that XPP.REG contains a snapshot of *HKEY_CURRENT_USER\Environment* as is before you install Xbase^{++}. If you change the registry later by installing other software, for example, these changes are not reflected in XPP.REG. Before you import XPP.REG into the registry, make sure that *HKEY_CURRENT_USER\Environment* is correct or remove this section entirely from the XPP.REG file.

1.5. Before you move on

If you have installed Xbase^{++} under Windows 95 or work with both Clipper and Xbase^{++} on the same workstation, please read the following sections before you start using Xbase^{++}.

1.5.1. Windows 95 environment limit

In Windows 95 the space for environment variables is limited to 1024 bytes by default. This can lead to problems when many paths are defined in AUTOEXEC.BAT pointing to numerous directories. If the memory space for environment variables is exhausted, it is most likely that paths are not defined completely or are not available in the system's environment at all. This is also the case if all paths are listed correctly in AUTOEXEC.BAT, since the environment table is full. When you encounter system error messages like "not sufficient environment space" you must increase the size of the environment table using the SHELL statement in your CONFIG.SYS file. The following line is an example that sets the environment space to 2048 bytes:

```
SHELL=C:\win95\command.com C:\win95 /E:2048 /P
```

Note that C:\win95 is only an example where the command processor COMMAND.COM is located. It can be in a different directory on your computer.

1.5.2. Working with Xbase++ and Clipper

Xbase++ needs system environment variables to be set correctly. This includes PATH, INCLUDE, LIB and XPPRESOURCE which are defined during default installation. However, if you work with Clipper on the same computer, there can be a conflict with the INCLUDE path. Both compilers search this path for include files. Also, both products use the same include file names such as STD.CH, for example, but the file contents differ to a great extent. Therefore, it is necessary that the INCLUDE path points to the correct include directory when you invoke either compiler.

A clear indication of a wrong include file being used is given when the Xbase++ linker reports an unresolved external function which begins with two underscores (for example: __SetFormat(), __Quit(), __Keyboard()). These function calls occur only if Clipper's STD.CH file is included. In general, there are no functions in Xbase++ which begin with a double underscore.

Switching the environment

In order to change the system environment settings for Xbase++ or Clipper more easily, two batch files are available after installation: *DINO.BAT* and *AUTOXPP.BAT*. The file DINO.BAT is located in the root directory of your C: drive and AUTOXPP.BAT is found in the root directory of the Xbase++ installation. DINO.BAT contains a pre-installation snapshot of the required environment variables and restores the system settings when invoked, while AUTOXPP.BAT sets the search paths for Xbase++. The two batch files, therefore, set the environment variables appropriate for Xbase++ or Clipper and let you switch the environment for either compiler on the fly.

2. After Installation

This chapter gives a short overview of how you can best get started working with Xbase^{++}. In addition, tips on how to obtain information about Xbase^{++} not included in this introductory documentation are given here.

2.1. Your first steps with Xbase^{++}

The easiest way to get a quick overview about Xbase^{++} and its features is to review sample programs that come along with Xbase^{++}. There is a variety of programming examples installed in the ..\SOURCE\SAMPLES directory which cover numerous different aspects. The example programs are divided up into four categories and are quickly reached by clicking the Start button -> Programs -> Alaska Xbase^{++} -> Xbase^{++} samples. This opens the Samples folder which again lists four folders, each of which contains samples of one category:

Apps	This category contains stand alone applications made up from multiple PRG source code files. Each application targets program design, like Single or Multiple Document Interfaces, for example.
Basics	Each example program in the Basics category focuses on one particular feature of Xbase^{++}. The samples are not intended for reuse but rather, show programming techniques which solve a particular problem. "How to display text using an italic font?", "How to create a modal window?" or "How to use a thread?" are questions answered by samples in this category.
Migrate	Migrating Clipper code to GUI is one of the strengths of Xbase^{++}. Therefore, samples are included in this category which show various possibilities and intermediate steps in the transition of a text mode application to a full featured GUI application.
Solution	This category collects example programs which provide solutions to common programming problems. All of them are stand-alone programs. However, they include program code which implement a test scenario for the actual example. The example is ready for reuse in your programs when you discard the code for the test scenario.

Examples in the Migrate folder provide a good starting point when you have installed Xbase^{++} for the first time. So open this folder and click on *Login*. Now you can actually see the contents of the directory ..\SAMPLES\SOURCE\MIGRATE\LOGIN which contains different file types. The file types are identified by the file extension and visualized by different icons.

2.1.1. Executing sample programs

Each sample directory includes the PRG source code and the corresponding executable file. Just double-click on the EXE file's icon to see what an example program does. You will find three EXE files in the *Login* folder: LOGIN_T.EXE, LOGIN_H.EXE and LOGIN_G.EXE. They all do the same (displaying a login dialog and a company logo), but they do it in different ways. LOGIN_T.EXE is a text mode application, LOGIN_H.EXE adds graphical elements to the text mode application and LOGIN_G.EXE is a GUI solution of the same problem. These three programs give you insight into some unique features of Xbase^{++}.

2.1.2. Viewing source code

When you have seen what an example program does, you might want to know how it is programmed. For this, you only need to double-click the PRG file's icon. This starts a text editor which loads the PRG file. By default, Windows' Notepad is started to display the PRG file. Note that each time you double-click a PRG file, a new instance of the editor might get started. If this is the case, you can drag a PRG file icon with the right mouse button and drop it on the editor. This way, the editor is not started again. Instead, it only loads the corresponding PRG file.

2.1.3. Running sample programs in the debugger

After studying the source code of an example program, you may still have questions about it. In this case, it is worth following the program by stepping through it in the debugger. You can start the debugger from the Start button -> Programs -> Alaska Xbase^{++} -> Xbase^{++} debugger. Once the debugger is up and running, you can drag an EXE file's icon with the right mouse button and drop it on the debugger window. Then you must click the OK pushbutton in the startup screen of the debugger to be able to step through the program with the F8 key. Note that you can invoke the debugger from the command line as well. This is described in a separate chapter later in this book.

2.1.4. Rebuilding sample programs

Once you have understood a sample program of your choice, you will probably want to play around with it and apply changes to see what happens if you change this or that in the PRG file. To see the results of changes, you must rebuild the corresponding EXE file. This is accomplished by double-clicking the icon of the PROJECT.XPJ file with the left mouse button. This invokes the Xbase^{++} ProjectBuilder which rebuilds the EXE (refer to the chapter The Xbase^{++} ProjectBuilder - PBUILD.EXE for more information).

2.2. The next steps with Xbase++

Once you have seen the example programs, you will have a good impression of the features of Xbase++. After this, where to go and how to proceed is up to you. You are well-advised to read the entire introductory documentation, but making further recommendations depends on your programming background and the language you have used so far. Xbase++ has an online documentation which is a huge source of information for answering almost any question. It is designed to be understood by a novice as well as an advanced programmer. Depending on your programming experience, you will find below some directions where to look in the online help for more and appropriate information. Maybe you can identify yourself with one of the following programmer types:

I've never used an xBase language

When you don't know anything about the xBase language, you should get familiar with its concepts. See the following chapters in the online help:

```
* Language Elements of Xbase++
* Elements of an Xbase++ Program
* Data Types and Literals
* Operators
* Declarations and Statements
* Operations and Operators for Simple Data Types
* Procedures, Functions and Special Operators
* Operations and Operators for Complex Data Types
```

I know the xBase language but have never used Clipper

If you are familiar with other xBase dialects such as FoxPro or dBase, for example, you should get a grasp about things possible in Xbase++. This includes the declaration of lexically scoped variables as well as data types such as Code block, Array and Object. You might be interested in the preprocessor too, since it can be used to translate your existing code to valid Xbase++ syntax. Have a look at these chapters in the online help:

```
* Declarations and Statements
* Operations and Operators for Simple Data Types
* Operations and Operators for Complex Data Types
* The Xbase++ Preprocessor
* Error Handling Concepts
```

Make yourself familiar with the functions and commands available in Xbase++. You will find them listed in alphabetical order when you open the Table of Contents tab in the online help.

I am a Clipper programmer

If you are a Clipper programmer, you should read the chapter "Information for Clipper programmers" in this book. It gives you head start information about migration issues.

I know Clipper but I don't know GUI

If you are a Clipper programmer who wants to migrate existing code to Windows, you should be familiar with the sample programs in the Basics and Migrate directory. Also, read the chapter "User Interface and Dialog Concepts" in this book. You should obtain a working knowledge of the following functions as well (you will find them in the index of the online help)

```
* AppEvent()
* PostAppEvent()
* SetAppWindow()
* XbpCrt()
* XbpDialog()
```

I have programmed Windows GUI applications already

You are familiar with the event-driven programming approach and you are not afraid of object-oriented programming. If this is the case, you should get familiar with the concept of Xbase Parts. They let you use GUI controls easily. Also, Xbase++'s multi-threading features are of interest for you. Read these chapters in this book:

```
* Basics of Xbase Parts
* Class hierarchy of Xbase Parts
* Multi-tasking and multi-threading
```

2.3. Online help system

Xbase++ includes a comprehensive Windows online help ..\BOOK\XPPREF.HLP. It is invoked via the Start button or by typing WINHLP32 XPPREF on the command line or by clicking the respective icon in the Xbase++ folder or the Windows explorer.

2.3.1. To search for information

If you are looking for a particular piece of information and don't know where to find it, try using the powerful search facilities provided by WINHLP32.EXE. The help file provides four ways to search for information:

Table of contents	The table of contents shows a hierarchical structure of all help panels. The first section Xbase++ Basics provides basic knowledge about features and programming concepts while other sections point to reference parts of the documentation. The Xbase++ Basics

	section is recommended for getting fundamental information about Xbase^{++}.
Index	The index tab page provides fastest access to a particular help topic. If you know the name of the function or command you are looking for, type it in this tab page and press the Return key.
Full text search	The full text search lets you search for a particular word in the entire documentation. All help panels which contain the specified word are listed. When this search option is used for the first time, WINHLP32.EXE asks for the creation of a word list for referencing help panels. This takes some time and creates the file XPPREF.FTS. Once this file is created, the full text search feature is available.
Quick reference	The quick reference is a special feature of the Xbase^{++} online help. It is invoked by clicking the QuickRef pushbutton in a help panel or by pressing Alt+Q. Then a window is displayed which lists help topics in alphabetical order.
	There are two Quick Reference windows. One lists functions and commands, while the other lists instance variables and methods. At the top of each Quick Reference window, a pushbutton is displayed which toggles both windows (Toggle QuickRef).

2.3.2. Accelerator keys in the help system

Accelerator keys are added to the Xbase^{++} online help to obtain fast access using the keyboard rather than the mouse.

Alt+C	Activates the Table of Contents tab page
Alt+Q	Opens the Quick Reference window
Alt+S	Accesses the Full Text Search
Alt+X	Activates the Index tab page
Alt+Y	Opens the History window
Ctrl+PgDn	Displays Next help topic
Ctrl+PgUp	Displays Previous help topic

All quick selections are also available in the context menu. It appears when clicking the help window with the right mouse button or by pressing the Ctrl+Return keys.

2.4. Xbase++ quality assurance

For quality assurance of its products, Alaska Software uses a defect tracking system for the following up and fixing of known bugs and anomalies. The "defect database" is used to collect information reported by registered users and route the information directly to the concerned department or even a single developer's desk. This allows for a short response time between bug report and bug fix.

2.4.1. How to contact Alaska

If the online help cannot answer your questions, our technical support will be glad to assist you and help solve your problems. Please use one of the following possibilities to contact our technical support:

E-Mail address for customers in Europe, Middle East, Africa:

support@de.alaska-software.com

E-Mail address for customers in America, Asia, Pacific, Australia:

support@us.alaska-software.com

CIS Forum:	GO ALASKA
FAX:	(+49) 6196/95 72 22
Phone:	(+49) 6196/95 72-0

2.4.2. How to report problems

Problem Description Reports (PDR) are used to report problems to Alaska. If you find an anomaly in the product, please use the form on the next page to include all possible information about the defect and fax it to Alaska Software. You can also use the PDR.TXT file which is located in the root directory of your Xbase++ installation and send it via eMail to our technical support. Sending an eMail is the preferred way because the PDR.TXT form can automatically be transferred into the defect tracking system.

2.4.3. The PDR form

```
##############< PDR status - this is filled in by ALASKA >#########

   PPN.: _________     PDR ID: ___________     REF No: _____________

Status : _________   Severity: ___________

########################< Customer data >#########################

Customer RegNo.: __________  CIS ID: __________   Date: __.__.____

 Version: 1.1.153 / Windows

Company / Sender : _______________________________________________

Person to contact: _______________________________________________

########################< Categories >############################

Online help  [ ]        Function/Command [ ]        Xbase Part [ ]
Preprocessor [ ]        Object/Method    [ ]        Graphic    [ ]
Compiler     [ ]        Database         [ ]        Other      [ ]

Symptoms        : _______________________________________________

Attached file(s): _______________________________________________

Remarks         : _______________________________________________

#################< Comprehensive description >####################




###########################< End >################################
```

2.4.4. What happens with PDRs

When you report a problem, it will be classified. If the problem is unknown so far and can be reproduced, it will be approved by the technical support. The problem then gets assigned a PDR-ID which identifies the problem for follow-up.

You will be notified by eMail if your reported problem is approved. Technical support will then provide you with information about how to work around the problem, if this is possible. All approved PDRs will be published frequently (approx. on a bi-monthly basis) in the ALASKA forum on CompuServe.

Note: We greatly appreciate any contribution that helps to improve the product and its robustness. However, if you want to contribute your experiences, you must be a registered user. Without a registration number, a PDR-ID cannot be assigned to a problem report.

3. Deployment of Xbase++ applications

Executable programs created with Xbase++ may be installed on other computer systems according to the license agreement you have with Alaska Software. Installation requires the EXE file plus Xbase++ runtime libraries to be copied to another workstation. The runtime libraries are contained in DLL files which are located in the RUNTIME directory of your Xbase++ installation. Which DLL file to copy depends partly on which DatabaseEngine is used by the EXE. The following table lists all files possibly required:

Files necessary to run Xbase++ applications

File		Description
XPPRT1.DLL	1)	Main runtime library of Xbase++
XPPUI1.DLL	1)	Events and Xbase Parts
XPPNAT.DLL	1)	Country-specific library (nation module)
SOM.DLL	1)	IBM SOM 2.0 runtime library
XPPUI2.DLL	2)	Application Parts and XbpBrowse
DBFDBE.DLL	2)	DatabaseEngine for DBF files
FOXDBE.DLL	2)	DatabaseEngine for FoxPro compatible DBF files
DELDBE.DLL	2)	DatabaseEngine for files in Delimited format
SDFDBE.DLL	2)	DatabaseEngine for files in SDF format
CDXDBE.DLL	2)	DatabaseEngine for CDX files
NTXDBE.DLL	2)	DatabaseEngine for NTX files

1) compulsory 2) optional

All DLL files marked with 1) in the table must be copied together with the EXE to another workstation. Otherwise the executable progam cannot be loaded into memory. The files marked with 2) are only necessary if the functionality of the respective DLL is requested by the EXE. The DLL files must be copied to a directory which is listed in the PATH environment variable. If the directory is not part of PATH, you must modify this environment variable accordingly.

Note: The table above lists all files of Xbase++ that are free for distribution to your customers. All files created by Xbase++ are also free for distribution (Please comply with the license agreement).

4. Information for Clipper Programmers

This chapter provides information for programmers planning to port existing Clipper applications to Xbase^{++}. Although Xbase^{++} is based on the Clipper language, certain differences exist due to compiler technology and differences between DOS and a 32bit operating system.

4.1. Differences in the preprocessor and compiler

The results of the Xbase^{++} and Clipper preprocessors and compilers differ in some very special situations. The following code examples show some situations where the Clipper preprocessor or compiler deviates from the Xbase^{++} preprocessor or compiler.

Command options

Using reserved function names as options in user-defined commands can lead to problems with the preprocessor in Xbase^{++}. For example:

```
#command @ <row>, <col> MYCOMMAND <expr> [MIN <min>] [MAX <max>] ;
      => MyFunction( <row>, <col>, <expr>, <min>, <max>)
```

This command is translated correctly unless the function Min() or Max() is used for the optional parameters <min> and <max>. The following would not be permitted in Xbase^{++}:

```
@ 10,20 MYCOMMAND "DoSomething" MIN Min(10,x)
```

In this case, the keyword used as part of the option of the command is identical to the keyword for the optional parameters. This is not permitted in Xbase^{++}.

Numeric constants

Numeric constants can be programmed in Xbase^{++} using decimal, hexadecimal or scientific notation. Clipper understands only decimal notation. The following constants are valid in Xbase^{++}:

```
  3.1415926      // decimal
  0xFF           // hexadecimal
  10.1E-10       // scientific
```

PARAMETERS statement

With the PARAMETERS statement, the Xbase++ compiler is more strict than Clipper. The statement is used to declare formal parameters for functions or procedures as variables of the PRIVATE storage class (Clipper '87). Since PARAMETERS is an executable statement, it may only be used **after** lexical variable declarations like LOCAL or STATIC. For example:

```
// Permitted                   // Not permitted
    PROCEDURE MyProc              PROCEDURE MyProc
       LOCAL cString                 PARAMETERS p1, p2
       PARAMETERS p1, p2             LOCAL cString

       <Code>                        <Code>
    RETURN                        RETURN
```

Note: In Xbase++, formal parameters should not be declared using PARAMETERS. Instead, a comma-separated list of parameters should be written within parentheses when declaring procedures, functions or methods:

```
PROCEDURE MyProc( p1, p2 )
       LOCAL cString

       <Code>
    RETURN
```

Alias operator

In Xbase++, only one variable identifier or the macro operator may be specified after the alias operator for dynamic memory variables M-> or MEMVAR-> and field variables FIELD->. An expression, with or without parentheses, cannot be used in these cases.

```
MEMVAR->(&cField)              // is supported in Clipper,
FIELD->(&cField)               // but not in Xbase++
```

Correct in Xbase++:

```
MEMVAR->&(cField)
FIELD->&(cField)
```

Macro operator &

When the macro operator is used within code blocks, Xbase++ always employs what is called "late evaluation". This can lead to different results in special cases. For example:

```
LOCAL  oTBrowse, oTBColumn, i
PRIVATE cFieldName

USE Customer NEW
oTBrowse:= TBrowseNew()

FOR i:=1 TO FCount()
   cFieldName := FieldName(i)
   oTBColumn  := TBColumnNew( cFieldName, {|| &cFieldName } )
   oTbrowse:addColumn( oTBColumn )
NEXT
```

The intention of this program code is to create a TBrowse object that displays columns for all fields in a database. The field name is assigned to a PRIVATE variable that is macro-expanded within the code block. In Clipper, "early evaluation" occurs and the code block references the field whose name is contained in the PRIVATE variable *cFieldName* when the code block is created. In Xbase++, the macro expression is only evaluated when the code block is evaluated. Because of this, all columns in the TBrowse object display the contents of the last field, since its name is contained in the variable *cFieldName* after the FOR...NEXT loop terminates. The following would be the correct code to use in this situation under Xbase++:

```
   oTBColumn  := TBColumnNew( cFieldName, &("{||"+cFieldName+"}") )
```

In this case, "early evaluation" is forced because the code block is first created as a character string and then macro-expanded. The code block then contains a reference to the actual field variable and not to the PRIVATE variable *cFieldName.*

The creation of file names with consecutive numbers as extension can be solved in Clipper in the following way:

```
PRIVATE nNumber := "1"

RESTORE FROM SaveFile.&nNumber
```

This macro expression is not valid in Xbase++. It must be recoded, for example by first assigning the file name to a variable and then using the variable as command parameter:

```
PRIVATE nNumber   := "1"
PRIVATE cFileName := "SaveFile.&nNumber"

RESTORE FROM &cFileName
```

If a syntax error is reported when a Clipper program is compiled in Xbase^{++}, the Xbase^{++} preprocessor output should be checked to see if it has generated the correct code. The output of the preprocessor can be written to a PPO file using the compiler option /p.

```
USE &(dictionary->cFilename)
```

The line above can be used in Clipper to open a file (cFilename) whose name is stored in a database (dictionary). The Xbase^{++} preprocessor converts &(...) to a character string. The correct code for Xbase^{++} is:

```
USE (dictionary->cFilename)
```

Reference operator @

Clipper allows the following:

```
x := IIf( y, @param, z )
```

When this code is compiled and executed under Clipper, x contains a reference to param if y equals .T. (true). This is not the case in Xbase^{++}.

On the other hand, with Xbase^{++}, any variable can be passed by reference to functions, procedures or methods. This includes array elements, member variables and field variables. The following function calls are allowed with Xbase^{++} but not with Clipper:

```
MyFunc( @Customer->Name )
MyFunc( @oTBrowse:colorSpec )
MyFunc( @aArray[3,12,5] )
```

#define __XPP__

The Xbase^{++} compiler defines the constant __XPP__. This allows the maintenance of different source code within a single PRG file when it is to be compiled by Clipper and Xbase^{++}.

```
#ifdef __XPP__
   <Xbase++ Code>
#else
   <Clipper Code>
#endif
```

#pragma

The Xbase^{++} compiler knows pragmas which do not exist in Clipper. A pragma is a directive that toggles compile switches at compile time. This allows a particular compile switch for a single line in a PRG file to be set or unset.

4.2. Differences between Clipper '87 and Xbase^{++}

Optimization of logical expressions: When compiling Clipper '87 programs Xbase^{++} uses the same short-cut optimization for logical expressions like Clipper 5.x:

```
IF <ExprA> .AND. <ExprB>
   // <ExprB> is not executed if <ExprA> == .F.
ENDIF

IF <ExprA> .OR. <ExprB>
   // <ExprB> is not executed if <ExprA> == .T.
ENDIF
```

Use the compiler switch /z to disable short-cut optimization when compiling Clipper '87 programs with Xbase^{++}. As an alternative the short-cut optimization can be activated|deactivated for a single line of code using *#pragma Shortcut(OFF|ON).*

Alias names: With Clipper '87, alias names consisting only of a single letter can be specified in the USE command. With Xbase^{++}, this is only permitted for letters above "M". For example, the following lines cannot be used with Xbase^{++} since they raise a runtime error:

```
USE Test1 ALIAS A
USE Test2 ALIAS K
```

Instead, the correct program code for Xbase^{++} would:

```
SELECT A
USE Test1
SELECT K
USE Test2
```

In general, it is recommended to use alias names with more than one letter.

4.3. Differences between Clipper 5.x and Xbase++

The most important aspect when porting a Clipper program to Xbase++ is given by the fact that the first routine of an Xbase++ application must be called MAIN. Without a MAIN procedure, the linker cannot determine the entry point in the program and the executable file cannot be loaded.

Program code located outside a FUNCTION or PROCEDURE declaration is automatically associated with the MAIN procedure by the Xbase++ compiler. Because of this, there can be only one PRG file with program code located outside FUNCTION or PROCEDURE declarations. Otherwise the MAIN procedure appears to be declared multiple times and an executable file is not created.

The second point to keep in mind is that Xbase++ programs always run in multiple threads. The thread executing implemented program code has the highest priority. It receives CPU access on preference by the operating system. This is why permanently polling DO WHILE loops should be avoided. For example:

```
DO WHILE Inkey() <> 0
   <program code>
ENDDO
```

Querying the keyboard in this way leads to permanent polling since there is no wait state in the loop. This is allowed under DOS but contradicts the architecture of a pre-emptive operating system. The above loop must be coded with Xbase++ as follows:

```
DO WHILE Inkey(0.1) <> 0
   <program code>
ENDDO
```

Now, the loop has a wait state of 1/10th of a second. This is sufficient to guarantee that all threads of a program will be executed. A Clipper program should be searched for DO WHILE loops which do a permanent polling and provide no wait state. Some functions accept a parameter to achieve a wait state. Another possibility for this is the *Sleep()* function.

The following table gives an overview of the commands that can lead to incompatibilities when a Clipper program is compiled using Xbase++.

Differences in commands

Clipper	Xbase++
CALL	Not available
COPY..TO..SDF	Xbase++ uses SDF as file extension for a structure extended SDF file
DIR	Not available
LABEL FORM	Not available
REPORT FORM	Not available
SET FORMAT	Not available
SET FUNCTION	Not available
SET TYPEAHEAD TO 0	Not allowed (min 10, max 100)
RESTORE FROM	Existing MEM files cannot be read
RUN	Starts a new command shell
FRM file	The FRM file format is not supported
LBL file	The LBL file format is not supported
MEM file	The MEM file format is not supported

When using commands to import or export data the features of a corresponding Xbase++ DatabaseEngine (DBE) must be taken into consideration. DBEs are loaded implicitly when commands are used. This applies to commands like COPY TO ... SDF or APPEND FROM ... SDF. The import or export file, respectively, is maintained by a DatabaseEngine which is loaded into memory if it is not found. In the case of the SDF format this is the SDF Datebase Engine. It creates a structure-extended file with the extension 'SDF'. For this reason, the following code is not allowed:

```
USE Customer
COPY TO Customer.sdf SDF
```

In this case, the Xbase++ SDFDBE implicitly creates the structure-extended file 'Customer.sdf'. It has the same name as the target file. As a result, a runtime error is raised. At this point, it is recommended to all Clipper programmers to read the specifications of the Xbase++ DatabaseEngines in the basic chapters of the Xbase++ documentation. The Xbase++ DBEs differ in many aspects from Clipper's RDDs.

Xbase++ differs from Clipper in that keyboard entries are registered in the event queue. The command SET TYPEAHEAD TO 0 in Xbase++ causes all events in the event queue to be deleted. The number of events in the event queue cannot be set to 0, since this would lead to a system halt. The minimum number of events that can be stored in the event queue is ten.

The MEM file format is not supported by Xbase++. This means existing MEM files cannot be read. In Xbase++, XPF files replace MEM files. Objects, arrays and code blocks can be stored using this file format as well as character, date, numeric and logical data types. This opens a completely new dimension for communicating between workstations on a network. Using the XPF file format, code blocks can be exchanged between two workstations.

Differences in functions

Clipper	Xbase++
AltD()	Not available
AEval()	Fifth parameter determines whether array elements are passed by reference to the code block
CurDir()	Returns or changes the current directory
Not available	CurDrive() Returns or changes the current drive
File()	Includes second parameter "D", "H", "S"
FkLabel()	Not available
FkMax()	Not available
NoSnow()	Not available
ReadKey()	Not available
SetBlink()	Only available in full screen mode (VIO mode)
SetColor()	Intensity attribute is supported for background color
Word()	Not available

In Xbase++, the function AEval() accepts a fifth parameter that is a logical value. This specifies whether array elements are passed by reference or by value to the code block. The following line creates an array and fills the 10 elements with their corresponding indexes:

```
aArray := AEval( Array(10), {|x,i| x:=i },,, .T.)
```

The functions CurDir() and File() have expanded functionality in Xbase++ (compared to Clipper). The additional function CurDrive() is also included in Xbase++. The current directory and drive can be changed using CurDir() and CurDrive() without using the command RUN. Since RUN operates differently under Xbase++ by starting another command shell, the Clipper technique will not change the directory in the application under Xbase++. The function File() accepts a file type as a second parameter. An example of the enhanced file function is shown in the following code that checks for the existence of a directory:

```
cDir := Space(64)
@ 10, 10 GET cDir
READ
```

```
IF ! File( Trim(cDir), "D")
   Alert( "Invalid directory")
ELSE
   CurDir( cDir )
ENDIF
```

The GetDoSetKey() function programmed in GETSYS.PRG calls GetPostValidate() prior to evaluating a SetKey() code block. By this, Xbase++ guarantees that no invalid data is written to database fields.

Reserved keywords

Xbase++ has more reserved keywords than Clipper. Using an Xbase++ reserved keyword as an identifier for variables, functions etc., causes the Xbase++ compiler to assert an error. A list of all reserved keywords is found in the online help in the section "Xbase++ basics", "Language elements of Xbase++", "Keywords".

Differences in instance variables and methods

Clipper	Xbase++
oGet:assign	oGet:_assign()
oGet:end	oGet:_end()
oGet:badDate	oGet:badDate()
oGet:minus	oGet:minus()
oGet:picture	When not indicated, default picture
oGet:subScript	Always NIL
Not available	oGet:posInBuffer()

oTbrowse:end()	oTBrowse:_end()
	Parameters for navigation code blocks:
	EVAL(oTBrowse:skipBlock, nSkip, oTBrowse)
	EVAL(oTBrowse:goTopBlock, oTBrowse)
	EVAL(oTBrowse:goBottomBlock, oTBrowse)

oTBrowse:rowPos:=<n>	Does not synchronize with the data source but only moves the TBrowse cursor
oTBrowse:nTop:=<n>	:configure() must be subsequently executed
oTBrowse:nLeft:=<n>	:configure() must be subsequently executed
oTBrowse:nBottom:=<n>	:configure() must be subsequently executed
oTBrowse:nRight:=<n>	:configure() must be subsequently executed
Not available	oTBrowse:firstScrCol()
Not available	oTBrowse:viewArea()

A leading underscore is added to the names of the *:assign()* and *:end()* methods of the Get and TBrowse classes in Xbase++. This is done because the previous identifiers are reserved keywords. No changes need to be made in Clipper programs, since the Xbase++ preprocessor makes this change.

Clipper requires the instance variable *oGet:subScript* to obtain correct references to array elements used as variables in the @...GET command. Xbase++ uses the reference operator @ instead, since single array elements can be passed by reference. For this reason, *oGet:subScript* is obsolete and always contains NIL.

There is an essential difference between the TBrowse class in Xbase++ and Clipper. Assignments to the instance variable *:rowPos* do not automatically synchronize the record pointer in the data source. If the TBrowse cursor is repositioned by changing *:rowPos*, the record pointer of the data source must be explicitly moved. Also, the TBrowse object is passed as a parameter to the code blocks *:skipBlock*, *:goTopBlock* and *:goBottomBlock*.

In addition, methods have been added to the Get class and the TBrowse class which are required for supporting mouse control.

Differences in database commands and functions

Clipper	**Xbase++**
DbSetDriver()	DbeSetDefault()
RddList()	DbeList()
RddSetDefault()	DbeSetDefault()
SET EXCLUSIVE	
Default value is ON	Default value is OFF
SET EXACT	
Is not considered with SEEK / DbSeek()	Is considered with SEEK / DbSeek()
CREATE INDEX	
Sorting order of characters depends on NTX???.OBJ modules that must be linked	Sorting order of characters depends on SET COLLATION and SET LEXICAL

Clipper	Xbase++
Structure extended file FIELD_LEN = 3	FIELD_LEN = 5
Field_len > 999 is coded using the field Field_dec	Field_dec is not considered for the length of a character field
COMMIT at end of program Is implicitly executed	Is not implicitly executed

RDD versus DBE

The database function DbSetDriver() and all the Rdd...() functions of Clipper are not available in Xbase++. This is because Xbase++ does not support the proprietary concept of a monolithic RDD (Replaceable Database Driver). Instead of the RDD approach, the concept of database engines is used in Xbase++ and offers much more flexibility in data and file management. Because of this difference, Xbase++ offers the Dbe...() functions instead of the Rdd...() functions in Clipper.

SET EXCLUSIVE

SET EXCLUSIVE is set OFF by default in Xbase++. This means that by default, databases are opened in SHARED mode for multi-user (network) operation in Xbase++. If a Clipper application not designed for multi-user access is recompiled under Xbase++, the command SET EXCLUSIVE ON needs to be inserted in the startup routine before any databases are opened.

SET EXACT

When SEEK and DbSeek() are used to search for character values in a database, Xbase++ uses the toggle SET EXACT to determine whether blank spaces at the end of character strings should be ignored in comparing character values.

CREATE INDEX

Xbase++ supports collation tables. A collation table assigns weighing factors to single characters. This allows the sorting order of characters to be user-defined. A collation table is used for string comparison as well as for index creation. The collation table is extremely important when creating index files to be accessed from both Xbase++ and Clipper. In this case, Xbase++ must use the same collation table as Clipper. In Clipper, the sorting order of characters is defined at link time by a nation module, and cannot be changed at runtime.

Clipper includes various NTX???.OBJ files that must be linked to the EXE in order to define the country-specific sorting of characters. To achieve the same sorting order with Xbase++, the corresponding collation table must be activated using the SET COLLATION TO command. The following table lists all language specific differences between Clipper and Xbase++.

Language-specific sorting of characters

Language	Clipper module	Xbase++ command
American	not available	SET COLLATION TO AMERICAN
British	not available	SET COLLATION TO BRITISH
Danish	NTXDAN.OBJ	SET COLLATION TO DANISH
Dutch	NTXDUT.OBJ	SET COLLATION TO DUTCH
Finnish	NTXFIN.OBJ	SET COLLATION TO FINNISH
French	NTXFRE.OBJ	SET COLLATION TO FRENCH
German	NTXGER.OBJ	SET COLLATION TO GERMAN
Greek 437	NTXGR437.OBJ	SET COLLATION TO GREEK437
Greek 851	NTXGR851.OBJ	SET COLLATION TO GREEK851
Icelandic 850	NTXIC850.OBJ	SET COLLATION TO ICELANDIC850
Icelandic 861	NTXIC861.OBJ	SET COLLATION TO ICELANDIC861
Italian	NTXITA.OBJ	SET COLLATION TO ITALIAN
Norwegian	NTXNOR.OBJ	SET COLLATION TO NORWEGIAN
Portuguese	NTXPOR.OBJ	SET COLLATION TO PORTUGUESE
Spanish	NTXSPA.OBJ	SET COLLATION TO SPANISH
Swedish	NTXSWE.OBJ	SET COLLATION TO SWEDISH
System	not available	SET COLLATION TO SYSTEM
ASCII		SET COLLATION TO ASCII

The default for SET COLLATION TO is defined in DBESYS.PRG. It depends on the country specific version of Xbase++. If index files are to be accessed from Xbase++ as well as from Clipper, you must include the following lines in your code:

```
#ifdef __XPP__
   SET COLLATION TO <your country>
#endif
```

Accessing index files from Xbase++ and Clipper at the same time requires both using the same sorting order for characters. When a Clipper program uses a different sorting order than an Xbase++ program, index files will get corrupted sooner or later.

CREATE FROM

The format of a structure-extended DBF file is different in Xbase++. The length of the field FIELD_LEN is 5, not 3 as in Clipper. This allows the maximum possible field length to be entered directly. It is no longer necessary to use the decimal place field to enter field lengths longer than 999 characters.

Memo Files

The maximum number of characters that can be stored in a memo field is not limited to 64 KB under Xbase++. If memo fields are accessed in a multi-user environment from Clipper and Xbase++, the 64 KB limitation must be implemented in Xbase++.

COMMIT at program's end

When a Clipper program ends, it commits all pending record updates to databases still open (the COMMIT command is implicitly executed). This is not the case with Xbase++. Instead, Clipper's behaviour is programmed in the file APPEXIT.PRG which contains the implicit EXIT PROCEDURE AppExit(). This procedure is called when a program terminates and can be changed to meet a programmer's needs.

RETURN and QUIT

An optional numeric parameter can be specified for QUIT. It is passed to the ErrorLevel() function before an application terminates. The same can be achieved if the function Main() returns a numeric value with the RETURN statement.

Other differences

Clipper	Xbase++
No Main procedure	PROCEDURE Main must be the first procedure of the application
Implicit INIT procedures in:	
	APPSYS.PRG
ERRORSYS.PRG	ERRORSYS.PRG
RDDSYS.PRG	DBESYS.PRG
Implicit EXIT procedures in:	
Not available	APPEXIT.PRG
PICTURE function @E	Numbers are displayed in the country specific format of the operating system
Transform(1.23,"@N")	Decimal point and thousands separator are configurable
Time()	Separators between HH:MM:SS are configurable
not available	SetLocale() configures country specific settings

Clipper	Xbase++
Constants for Set() function	
_SET_DEBUG _SET_SCROLLBREAK	These constants are not supported
New constants for Set() not available	_SET_CHARSET _SET_COLLATION _SET_LEXICAL _SET_TIME

The configuration settings of the operating system are used to determine the default separators used in formatting numeric and date values as well as the return value of the function Time(). Thus, the PICTURE formatting with @E is obsolete and is ignored by Xbase++. However, numerous country specific settings can be configured with the function SetLocale().

4.4. New functions in Xbase++ (unknown in Clipper)

This section lists all functions, commands and statements which are new in Xbase++ compared to Clipper.

New functions in Xbase++

Name	Description
AppDesktop()	Returns the desktop window
AppEvent()	Reads events from the event queue
AppType()	Determines the application type
Bin2u()	Converts a binary string to a LONG integer
Bin2Var()	Converts a binary string to any data type
ClassCreate()	Creates a class dynamically at runtime
ClassDestroy()	Releases a dynamic class object
ClassObject()	Returns the class object of any class
ConfirmBox()	Displays GUI dialog box for user confimation
ConvToAnsiCP()	Converts a string to ANSI
ConvToOemCP()	Converts a string to OEM
CurDrive()	Returns current drive letter
DispOutAt()	Screen output without cursor position change
DllCall()	Calls a function contained in a DLL
DllExecuteCall()	Calls a DLL function using a call template

Name	Description
DllLoad()	Loads a DLL dynamically at runtime
DllPrepareCall()	Prepares a call template for a DLL function
DllUnLoad()	Unloads a DLL
EnableClipRect()	Enables clipping for virtual text mode screen
GetEnableEvents()	Enables the mouse for GETs
GetEventReader()	GETs use AppEvent() instead of Inkey()
GetHandleEvent()	Default event handler for GETs
GetKillActive()	Takes focus from active GET
GetList()	Returns current GetList array
GetListPos()	Returns position of current GET in GetList
GetToMousePos()	Moves the cursor to the mouse within GETs
IsFieldVar()	Checks if field exists
IsFunction()	Checks if function exists
IsMemberVar()	Checks if object has a specific member variable
IsMemvar()	Checks if memory variable exists
IsMethod()	Checks if object has a specific method
LastAppEvent()	Returns last event
MsgBox()	Displays GUI message box
NextAppEvent()	Returns next event
NumButtons()	Number of mouse buttons
PostAppEvent()	Posts an event to the event queue
PValue()	Retrieves value of n-th parameter
RunRexx()	Runs a REXX command file
RunShell()	Opens a new command shell
SetAppEvent()	Associates an event with a code block
SetAppFocus()	Sets focus to a window or GUI control
SetAppWindow()	Returns the application window
SetClipRect()	Defines the clipping area for text mode
SetLexRule()	Defines lexical comparison rules for characters
SetLocale()	Function for localization
SetMouse()	Enables mouse events in text mode
SetTimerEvent()	Starts a timer thread
Sleep()	Halts the current thread for specified time
TBApplyKey()	Default Inkey() handler for TBrowse
TBHandleEvent()	Default AppEvent() handler for TBrowse
TBtoMousePos()	Moves the TBrowse cursor to the mouse pointer
ThreadID()	Returns ID of current thread
ThreadObject()	Returns current Thread object
ThreadWait()	Waits for one thread to terminate
ThreadWaitAll()	Waits for multiple threads to terminate
U2bin()	Converts an unsigned LONG integer to binary
Var2Bin()	Converts any data type to binary

New Database related functions

Name	Description
DbCargo()	Attaches an arbitrary value to a used work area
DbClientList()	Returns all registered clients of a work area
DbContinue()	Functional equivalent of CONTINUE
DbCopyExtStruct()	Functional equivalent of COPY STRUCTURE EXTENDED
DbCopyStruct()	Functional equivalent of COPY STRUCTURE
DbCreateExtStruct()	Functional equivalent of CREATE STRUCTURE EXTENDED
DbCreateFrom()	Functional equivalent of CREATE FROM
DbDeregisterClient()	Removes a registered client from a work area
DbeBuild()	Builds a compound DatabaseEngine
DbeInfo()	Returns information about a DatabaseEngine
DbeList()	Returns the names of loaded DatabaseEngines
DbeLoad()	Loads a DatabaseEngine
DbeSetDefault()	Selects the default DatabaseEngine
DbeUnload()	Releases a DatabaseEngine from memory
DbExport()	Functional equivalent of COPY TO
DbGoPosition()	Moves the record pointer using a percent value
DbImport()	Functional equivalent of APPEND FROM
DbInfo()	Returns information about a work area
DbJob()	Associates a work area with a code block
DbJoin()	Functional equivalent of JOIN
DbList()	Functional equivalent of LIST
DbLocate()	Functional equivalent of LOCATE
DbPack()	Functional equivalent of PACK
DbPosition()	Returns the record pointer position as percent value
DbRegisterClient()	Registers a client in a work area
DbRelease()	Releases a work area from current work space
DbRequest()	Transfers a work area into current work space
DbResetNotifications()	Enables notifications from work areas to clients
DbSkipper()	Default DBF skipper function for TBrowse
DbSort()	Functional equivalent of SORT
DbSuspendNotifications()	Disables notifications from work areas to clients
DbTotal()	Functional equivalent of TOTAL
DbUpdate()	Functional equivalent of UPDATE
DbZap()	Functional equivalent of ZAP
FieldInfo()	Returns field information
OrdIsDescend()	Checks if index is descending
OrdIsUnique()	Checks if index is unique
WorkSpaceEval()	Evaluates a code block in all used work areas
WorkSpaceList()	Returns alias names of all used work areas

New Directives, Statements and Commands

Name	Description
#pragma	Toggles compiler switches at compile time
ACCESS | ASSIGN	Declares access/assign methods
APPBROWSE	Application part (GUI browser)
APPDISPLAY	Displays application parts
APPEDIT	Application part (Edit window)
APPFIELD	Declares a field for an application part
CLASS	Declares a class
CLASS METHOD	Declares a class method
CLASS VAR	Declares a class variable
DEFERRED	Declares method as deferred
DLLFUNCTION	Creates a function which calls a DLL function
FINAL	Declares method as final
INLINE	Declares inline method
METHOD	Declares method
SET CHARSET	Selects ANSI or OEM character set
SET COLLATION	Sets country specific collation table
SET LEXICAL	Enables lexical comparison rules for strings
SET TIME	Sets display format for the system time
SYNC	Declares method as synchronized
VAR	Declares an instance variable

Xbase^{++} Class functions

Name	Description
DataRef()	Class for referencing data
Error()	Error class
Get()	Get class
TBrowse()	TBrowse class
TbColumn()	TBColumn class
Thread()	Thread class
Signal()	Signal class
VCrt()	Virtual screen class for text mode
Xbp3State()	Three state button class
XbpBitmap()	Bitmap class
XbpBrowse()	GUI browser class
XbpCheckBox()	Checkbox class
XbpClipBoard()	Clipboard class
XbpColumn()	Column class for GUI Browser
XbpComboBox()	Combobox class

Name	Description
XbpCrt()	Hybrid window class
XbpDialog()	GUI window class
XbpFileDev()	File device class
XbpFileDialog()	File selection dialog
XbpFont()	Font class
XbpFontDialog()	Font selection dialog
XbpHelp()	Class for online help window
XbpHelpLabel()	Class for context sensitive help
XbpIWindow()	Implicit window
XbpListBox()	Listbox class
XbpMenu()	Menu class
XbpMenuBar()	Menubar class
XbpMetaFile()	Metafile class
XbpMLE()	Multi line edit class
XbpPartHandler()	Parent class for XbaseParts
XbpPresSpace()	Presentation space
XbpPrinter()	Printer class
XbpPushButton()	Pushbutton class
XbpRadioButton()	Radiobutton class
XbpScrollBar()	Scrollbar class
XbpSetting()	Abstract class for switchcs
XbpSLE()	Single line edit class
XbpSpinButton()	Spinbutton class
XbpStatic()	Static GUI control class
XbpSysWindow()	Abstract class for system dialogs
XbpTabPage()	Tabpage class
XbpTreeView()	Tree view class
XbpTreeViewItem()	Items in tree view
XbpWindow()	Abstract window class

Functions of the GraphicsEngine

Name	Description
GraArc()	Draws an arc, circle or ellipsis
GraBitBlt()	Copies a bitmap
GraBox()	Draws a rectangle
GraError()	Returns last error code
GraLine()	Draws a line
GraMarker()	Draws a marker
GraPathBegin()	Opens a graphic path

Name	Description
GraPathClip()	Uses graphic path for clipping
GraPathEnd()	Closes a graphic path
GraPathFill()	Fills a graphic path
GraPathOutline()	Outlines a graphic path
GraPos()	Sets the pen position
GraQueryTextBox()	Calculates coordinates for strings
GraRotate()	Rotates graphic output
GraScale()	Scales graphic output
GraSegClose()	Closes a graphic segment
GraSegDestroy()	Releases a graphic segment
GraSegDraw()	Draws a graphic segment
GraSegDrawMode()	Sets the draw mode for graphic segments
GraSegFind()	Finds graphic segments
GraSegOpen()	Opens a graphic segment
GraSegPickResolution()	Sets sensitivity for finding graphic segments
GraSegPriority()	Sets the priority for graphic segments
GraSetAttrArea()	Sets attributes for drawing areas
GraSetAttrLine()	Sets attributes for drawing lines
GraSetAttrMarker()	Sets attributes for drawing markers
GraSetAttrString()	Sets attributes for drawing strings
GraSetColor()	Sets the color for all graphic functions
GraSetFont()	Sets the font for drawing strings
GraSpline()	Draws a curve
GraStringAt()	Draws a string
GraTranslate()	Shifts graphic output

4.5. Differences between Class(y) and Xbase++

Class(y) is the leading third-party product for object-oriented programming under Clipper. The Xbase++ syntax for the object model is similar to Class(y), but does not copy the syntax because the object model and the implementations are different. The most important difference is that the Xbase++ compiler takes care of class declarations instead of the Clipper preprocessor, as used by Class(y). Because of how the preprocessor is used by Class(y), only one class can be declared in each PRG file. Xbase++ allows any number of classes to be declared in a single file. Class(y) translates each METHOD to a STATIC FUNCTION and the CREATE CLASS declaration to a FUNCTION. Xbase++ handles things differently. The compiler creates a class function from the CLASS declaration and creates references to method implementations, not STATIC FUNCTIONs. In Xbase++, a method differs from a static function because the variable *self* implicitly references the object executing the method. The variable *self* cannot be declared or changed within methods. Assigning a value to *self* or passing *self* to a function using the reference operator generates an Xbase++ compiler error.

An important instance variable in Class(y) is *::super*. This instance variable does not exist in Xbase++ since it is not appropriate for clearly identifying superclass objects in multiple inheritance scenarios. Xbase++ accesses the member variables and methods of each superclass using the name of the superclass. Example:

```
CLASS DataDialog FROM XbpDialog
   EXPORTED:
   METHOD init
   <...>
ENDCLASS

METHOD DataDialog:init( parameters )
   ::xbpDialog:init( parameters )
   <...>
RETURN self
```

The *:init()* method of the superclass must be called by this explicit "cast" in Xbase++. Any visible method in any superclass can be explicitly called by specifying the class name in the call to the method.

Another important difference between Class(y) and Xbase++ is the fact that class methods and class variables are only accessible through the class object under Class(y). Under Xbase++, each instance of a class has access to class methods and class variables. Calling a class method can occur via any object (instance) of a class in Xbase++. Class variables are also accessible through instances. Xbase++ does not require a MESSAGE...TO CLASS to be declared in the class to delegate class variable and method calls to the class object.

Since the Xbase++ syntax for declaring classes and methods is so similar to the Class(y) syntax, porting Class(y) code to Xbase++ is relatively easy with help from the preprocessor. As an example, the preprocessor directives included below translate Class(y) code to Xbase++:

```
/*  Class(y)                                 Xbase++   */
#ifdef __XPP__

   #command CREATE CLASS <x> [FROM <y>]   => CLASS <x> [FROM <y>]
   #command MESSAGE <x> METHOD <y>        => METHOD <x> IS <y>
   #command CLASS MESSAGE <x> METHOD <y> => CLASS METHOD <x> IS <y>
                                             // Name of the superclass
   #xtrans :super                         => :<superClassName>

#else
   #include "Class(y).ch"
#endif
```

If Class(y) code is to be ported to Xbase^{++}, the preprocessor can be used to accommodate differences in syntax. The #ifdef directive allows the same code to be compiled using either Clipper/Class(y) or Xbase^{++}. The PRG file containing the Class(y) class declaration just needs to be compiled using preprocessor directives that translate the existing code into valid Xbase^{++} code.

4.6. Using the mouse

The first step in porting existing Clipper programs to a 32bit operating system is integrating the mouse for program navigation. In Xbase^{++}, all dialog functions and commands can be controlled using the mouse. This includes @...SAY...GET, MENU TO, Alert(), TBrowse(), and Memoedit(). To provide mouse support, the function SetMouse(.T.) needs to be inserted into the start routine of the program. If this is included, the function AppEvent() is used to retrieve events instead of the function Inkey(), which reads only keyboard input. Both functions return a numeric value corresponding to either the key code of a pressed key (Inkey()) or the identity of the last event that occurred (AppEvent()). With Clipper, Inkey() is the function generally used to retrieve keyboard input. This function only exists in Xbase^{++} for compatibility. Xbase^{++} replaces Inkey() with the function AppEvent() in order to read events from the event queue.

The return values of AppEvent() and Inkey() may be different when the same key is pressed because the DOS Inkey() code is not compatible with the AppEvent() code. For this reason, the transition to mouse support using SetMouse(.T.) also requires that calls to SetKey() be replaced by calls to the function SetAppEvent(). Code blocks that are associated to keys using SetKey() must be associated to the same key using SetAppEvent(). The #define constant used to identify keys is also different. For example:

```
SetKey( K_F10, {|| DoSomething() } )          // Clipper

SetAppEvent( xbeK_F10, {|| DoSomething() } )  // Xbase++
```

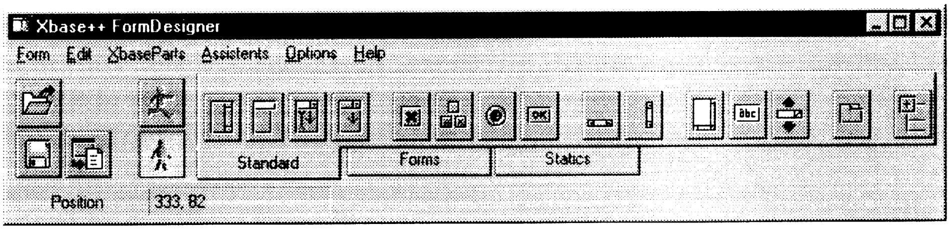
The first line contains Clipper code. This call associates the function key F10 to a code block. This code also works correctly in Xbase^{++} as long as the mouse is not activated. After SetMouse(.T.) is called to activate the mouse, the code block {|| DoSomething() } will not be executed when the F10 key is pressed. To guarantee execution of the appropriate code block after the F10 key is pressed, the code block must be associated to the xbeK_F10 event code rather than the K_F10 key code using SetAppEvent().

5. The Xbase⁺⁺ FormDesigner - XPPFD.EXE

The Xbase⁺⁺ FormDesigner is a powerful tool designed to support the programmer in the development of GUI applications (Graphical User Interface). The FormDesigner is written entirely in Xbase⁺⁺ which demonstrates the versatility of this development package. This chapter describes how to utilize and work with the FormDesigner and how generated code can be integrated into an application program.

5.1. Components of the FormDesigner

The FormDesigner is started by double-clicking the corresponding icon in the Xbase⁺⁺ folder or by entering XPPFD on the command line. The main window of the FormDesigner is initially displayed together with a blank form. The new form is an XbpDialog window into which GUI controls, or Xbase Parts, are inserted. The Xbase Parts are selected by clicking corresponding icons in the main window of the FormDesigner.



Main window of the FormDesigner

The main window is divided into three sections: menu bar, tool palette and status line. All functions of the FormDesigner can be selected via the menu bar. The major functions are integrated into the tool palette and are accessible by clicking corresponding icons with the mouse. For comfortable usage, the icons are contained in three tabbed pages.

The main window is complemented by supplementary windows which are opened via the menu system and not permanently visible.

The following table gives an overview:

Additional windows of the FormDesigner

Menu item	Window
Edit	
- Sequence	Changes the sequence of Xbase Parts in a form
- Symbols for iVar	Defines symbols for instance variables used in object-oriented code
Assistents	
- Fields	Selects database files and fields
Options	
- Settings	Defines default settings for a form and the FormDesigner
- Alignment pallette	Contains icons for default alignment of Xbase Parts within the marking rectangle
- Property monitor	Changes properties of Xbase Parts contained in a form
- Resolution monitor	Displays the form in comparison with different screen resolutions
Help	
- Help index (F1)	Displays the online help of the FormDesigner

Note: the FormDesigner has a context-sensitive online help that is activated by pressing the F1 key. To get detailed information about a particular window, the window must have focus before F1 is pressed.

5.2. Working with the FormDesigner

Working with the FormDesigner always involves following distinct steps: selection of controls or database fields, arrangement in the form, saving the form, and creation of program code. The easiest way for selection of controls or Xbase Parts, respectively, is clicking a corresponding icon in the tool palette of the FormDesigner. Then an insertion frame is displayed in the form which is to be positioned with the mouse. A left click marks the insert position and creates an Xbase Part in the form at this point. If database fields are to be inserted, the field selection window must be opened via the menu items "Assistents->Fields". Multiple field names can be selected in this window. Depending on the data type of a field, different Xbase Parts accessing a database field are created in the form. They must be positioned with the insertion frame.

The FormDesigner has two modes to position the insertion frame: pixel-oriented and grid-oriented. The positioning mode is selected in the settings dialog ("Use grid") which is opened via the menu items "Options->Settings". This dialog defines settings to configure the FormDesigner. One setting, for instance, is the grid size used for positioning. If the grid is activated, Xbase Parts can be moved in the form only in steps defined by the grid. To position an Xbase Part exactly at the mouse pointer tip, the grid must be deactivated.

While a form is designed, it is normally necessary to reposition inserted Xbase Parts, to group them using static elements such as lines or frames, or to make other changes. The form can be modified comfortably by means of a marking frame that allows the entire appearance of a form to be changed with just a few mouse clicks. The marking frame selects one or more Xbase Parts for modification. Position and size of the marking frame define position and size of the marked elements. If the frame is resized, different options are available to define how the marked Xbase Parts are to be scaled. These options are selected in the "Options->Settings" window. For instance, the spacing between Xbase Parts can be scaled, or their size, or both. Whether or not controls embedded in Xbase Parts (children) should be scaled as well, is also configurable. A variety of standardized alignment options can be selected in the "Alignment" window. All marked Xbase Parts can be centered in the marking frame or given the same width by just clicking the corresponding icon.

Parent-child relationships between Xbase Parts can be defined or modified easily. Various options exist for this purpose in the context menu of the form which is opened by a right mouse click.

Options in the context menu

Option	Description
Group XBPs	Creates a Group box as the parent for all marked Xbase Parts
Ungroup XBPs	Removes the Group box and defines the form as the parent for Xbase Parts contained in

Option	Description
	the Group box
Assign parent	Defines an Xbase Parts as the parent for all marked elements
Release parent	Releases a parent-child relationship without deleting the parent
To back	Moves all marked Xbase Parts to the background
To front	Moves all marked Xbase Parts to the foreground
Repaint all	Redraws the entire form

Xbase Parts are grouped using a Group box by default (an XbpStatic element of the type XBPSTATIC_TYPE_GROUPBOX). It is used as parent for marked Xbase Parts. After a group is defined in this way only the parent can be marked, not the embedded Xbase Parts. To select or modify children of an Xbase Part, it is necessary to release the parent-child relationship.

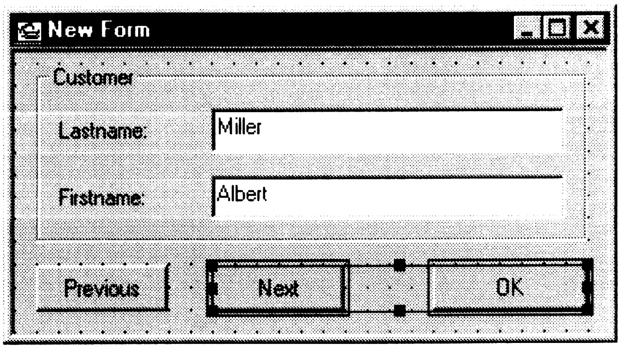
Clicking the menu items "Options->Property monitor" opens a window which is used to change proprties of Xbase Parts contained in the form. It uses a tree view for visualization of the parent-child hierarchy and is equipped with a browser which displays the properties of the current Xbase Part. If properties are changed in the browser the corresponding Xbase Part is updated with new data.

When a form is completely designed, it can optionally be stored or code can be generated in both functional or object-oriented style. This code can be modified further using a standard text editor. In general it is recommended to store a form in case it needs to be modified later on, or to serve as template for other forms. The FormDesigner stores a form in binary files which can be loaded at a later point in time. Which type of source code is generated by the FormDesigner is selected in the menu system ("Form->FUNCTION Code" or "Form->CLASS Code"). As an alternative, the default type is created by clicking an icon in the tool palette. The default source code type is selected in the Settings window.

5.3.FUNCTION Code

Before source code is generated by the FormDesigner, it is necessary to sort the Xbase Parts in the form. They must appear in the order that corresponds to the sequence a user accesses them by using the tab key. Normally the source code for Xbase Parts is created in the same sequence in which they are inserted into the form. This sequence may not be equivalent to user interaction. The sequence can be checked and/or modified in the Sequence window (menu items: Edit->Sequence). Source code should be created only if the sequence of Xbase Parts corresponds with user interaction.

A sample form can be found in the ..\SAMPLES\BASICS\XPPFD directory which was created by the FormDesigner. It is used as an example for further explanations and is shown below:



Example for a customer form

The form contains two grouped XbpSLEs to edit database fields. Pushbuttons are arranged below the Group box for database navigation and to close the form. Both types of source code for this form, functional and object-oriented, exist in the ..\SAMPLES\BASICS\XPPFD directory. It can be compiled and executed. For the source code to function correctly, the FormDesigner makes some simplifying assumptions that must be considered when the code is integrated into an application program.

The functional source code starts with all necessary *#include* directives and declares the Main procedure plus some LOCAL variables. Then all databases that are accessed by a form are opened with the USE command. The sample form uses only one database:

```
#include "Gra.ch"
#include "Xbp.ch"
#include "Appevent.ch"

PROCEDURE Main
   LOCAL nEvent, mp1, mp2
   LOCAL oDlg, oXbp, drawingArea, oXbp1, aEditControls := {}

   // Path is abbreviated only in the documentation
   USE ..\SAMPLES\BASICS\XPPFD\CUSTOMER.DBF NEW EXCLUSIVE
```

The LOCAL variable *aEditControls* is initialized with an empty array. It is used to collect all Xbase Parts that access database fields. It provides a comfortable way to access all Xbase Parts in a user-defined function that edit data in the form.

Since the FormDesigner cannot make any assumptions about possible record locking strategies in a multi-user scenario, the databases are opened for exclusive access. Therefore, the program lines that open databases with the USE command are most likely subject to change when the source code is integrated into an application program.

After the databases are opened, the form is created as an instance of the XbpDialog class and a reference to the drawing area is assigned to a LOCAL variable.

```
oDlg := XbpDialog():new( SetAppWindow():setParent(), , ;
                         {148,98}, {334,182}, , .F.)
oDlg:border := XBPDLG_SIZEBORDER
oDlg:title := "New Form"
oDlg:create()

drawingArea := oDlg:drawingArea
drawingArea:setColorBG( GRA_CLR_PALEGRAY    )
drawingArea:setFontCompoundName( "8.Helv.normal" )
```

These lines of code define parent, position and size of the form plus border type and title. Also, background color and font are set. This code most probably needs to be modified except for position and size. Especially the parent window should be checked if it is correct. The FormDesigner assumes the desktop window will be used as parent and writes the expression *SetAppWindow():setParent()* to retrieve it. This again relies on the assumption that an application window is defined in APPSYS.PRG which is different from the desktop window.

Note: If this code is to be executed inside the AppSys() procedure, only *SetAppWindow()* may be specified as parent. When a program starts, the desktop window is the application window. Since the desktop window has no parent, the expression *SetAppWindow():setParent()* leads to a runtime error inside AppSys().

The LOCAL variable *drawingArea* references the drawing area of the form. It is used as parent for subsequent Xbase Parts:

```
oXbp1 := XbpStatic():new( drawingArea, , {12,48}, {300,96} )
oXbp1:caption := "Customer"
oXbp1:type := XBPSTATIC_TYPE_GROUPBOX
oXbp1:create()
```

These four lines of code create the group box used to group the entry fields in the form. Since it serves as parent for the XbpSLEs, the XbpStatic object is assigned to a separate LOCAL variable *oXbp1*. This variable is used in the source code to define the parent for XbpSLE objects:

```
oXbp := XbpStatic():new( oXbp1, , {12,48}, {72,24} )
oXbp:caption := "Lastname:"
oXbp:options := XBPSTATIC_TEXT_VCENTER+XBPSTATIC_TEXT_RIGHT
oXbp:create()
```

```
oXbp := XbpSLE():new( oXbp1, , {96,48}, {192,24} )
oXbp:bufferLength := 20
oXbp:tabStop := .T.
oXbp:dataLink := {|x| IIf( PCount()==0, ;
                          Trim( CUSTOMER->LASTNAME ), ;
                                CUSTOMER->LASTNAME := x ) }
oXbp:create():setData()
AAdd( aEditControls, oXbp )
```

When Xbase Parts access database fields but have no caption, an XbpStatic object is always created to display text left to the entry field. The text defaults to the field name and is displayed right-aligned in mixed case.

For the program logic, the *:dataLink* code block is essential since it performs read and write access to a single field. In functional style source code, literals appear in the code block for alias and field name. Therefore, the code must be adapted, for example, when a different alias name is specified in the USE command.

The *:bufferLength* instance variable limits the number of characters that can be entered into the edit buffer of an XbpSLE object. It equals to the length of the corresponding database field, so an XbpSLE cannot edit more characters than can be stored in a field. Since blank spaces are valid characters for an XbpSLE, the *:dataLink* code block must remove trailing spaces when it reads a database field. Otherwise the edit buffer is already full after read access and no changes will be possible when the XbpSLE is in insert mode. Characters can only be overwritten in this case.

Note: in functional code, the LOCAL variable *oXbp* always receives the reference to an Xbase Part which is not used as parent. In the example above, it first references an XbpStatic object. After the *:create()* method is executed, a new object reference is assigned to this variable (oXbp := XbpSLE():new(...)). This does not overwrite or destroy the XbpStatic object. When the *:create()* method is finished, the XbpStatic object is already added to the child list of the parent, which is the Group box referenced in *oXbp1.* A reference to the XbpStatic object could be retrieved with the expression *oXbp1:childList()[1].* After the XbpSLE object is created, it is added to the *aEditControls* array. All Xbase Parts that access database fields via the *:dataLink* code block are added to this array and can be accessed in a program using this variable.

The source code generated for the creation of Xbase Parts can be quite extensive and may look complex at the beginning. However, it consists only of a simple pattern that is repeated:

```
oXbp := Xbp...():new( <parent>, , <position>, <size> )
oXbp:<configuration> := <value>
oXbp:create()
```

This pattern corresponds to the "life cycle" of Xbase Parts. An instance of a class is created with the *:new()* method. The instance (object) gets assigned configuration values. According to the configuration, system resources are requested from the operating system with the

:create() method. After *:create()*, the Xbase Part functions and is referenced in the child list of its parent. Therefore, the LOCAL variable *oXbp* can be reused.

The FormDesigner generates standardized source code and releases a programmer from a lot of 'typing work'. Source code for the layout and the creation of a form, or for accessing database fields (*:dataLink*), can be standardized. The limit of the FormDesigner is reached, however, when it comes to program flow or logic. This cannot be standardized and it remains a programmer's task to adapt generated source code in order to integrate it into an application program.

But in programming there are situations common to many applications. Moving the record pointer is an example. For this, a set of predefined pushbuttons can be selected in the FormDesigner and inserted in the form. The following lines of code are an example for a pushbutton that moves the record pointer to the next record:

```
oXbp := XbpPushButton():new( drawingArea, , {12,12}, {72,24} )
oXbp:caption := "Next"
oXbp:create()
oXbp:activate := {|| Gather( aEditControls ), ;
                     CUSTOMER->( DbSkip( 1) ), ;
                     Scatter( aEditControls ) }
```

In this case, the program is controlled by the *:activate* code block which is evaluated when the pushbutton is clicked. The function Gather() writes the current values from the form into the database. Then, the record pointer is moved by the DbSkip() function, and finally values from the new record are transferred to the form with the Scatter() function. The new values are then visible on the screen.

The most important aspects of the FormDesigner can be summarized as follows:

1. The FormDesigner generates source code based on the visual representation of a form.
2. The sequence of Xbase Parts in a form must be adjusted to correspond with user interaction.
3. Source code that controls program flow is created to a very limited extent. There is only a set of pushbuttons which perform common tasks with their *:activate* code block.
4. The generated source code must be modified when integrating it into an application program.

5.4. CLASS Code

The Xbase^{++} FormDesigner is able to create source code for a form in object-oriented style. This includes the declaration of a class (whose instances display the form) plus code for the *:init()* and *:create()* methods. For the creation of object-oriented code, the same rules are valid as for functional code (refer to the previous section). The sequence of Xbase Parts in the form must be checked. In addition, symbols for instance variables of the class should be defined. If they are not defined, the FormDesigner creates default identifiers for instance variables from the class name of an Xbase Part plus a digit, or it uses field names as symbols when an Xbase Part accesses a database field. In order to get code with good readability, the names (symbols) of instance variables should be edited prior to generating the source code. This is accomplished in the symbols window (menu items: Edit->Symbols for iVar).

The object-oriented code takes advantage of inheritance. Two PRG files are created, each of which contains the code for one class. The first file contains the implementation level of the form while the second one provides the utilization level. The code is split into two files and one class (utilization level) is derived from the other one (implementation level). For the example form, discussed in the previous section, two files exist in the ..\SAMPLES\BASICS\XPPFD directory: SAMPLE2.PRG and _SAMPLE2.PRG. Both contain code to create one class.

```
Implementation level
--------------------
   File name  : _SAMPLE2.PRG
   Declaration: CLASS _CustomerForm FROM XbpDialog

Utilization level
-----------------
   File name  : SAMPLE2.PRG
   Declaration: CLASS CustomerForm FROM _CustomerForm
```

For the creation of the sample code the file name SAMPLE2.PRG and class name *CustomerForm* were used. The FormDesigner prefixes both names with an underscore and uses the resulting identifiers for the source code that builds the implementation level of a form. The implementation level class has instance variables for each Xbase Part contained in the form. In addition, instance variables for each accessed database plus the instance variable *:editControls* are declared. The source code of the sample form for the implementation level is given below:

```
CLASS _CustomerForm FROM XbpDialog
   EXPORTED:
   VAR editControls

   * Contained control elements
```

```
    VAR GroupBox
    VAR Static1
    VAR Lastname
    VAR Static2
    VAR Firstname
    VAR ButtonNext
    VAR ButtonPrevious
    VAR ButtonOK

    * Work area / alias
    VAR CUSTOMER

    METHOD init
    METHOD create
ENDCLASS
```

In addition to instance variables, the methods *:init()* and *:create()* are declared. They initialize the form and request system resources for it. The source code for these methods is created by the FormDesigner. Both methods have the same parameter profile as other Xbase Parts. Default values for all parameters are defined in the *:init()* method.

```
METHOD _CustomerForm:init( oParent, oOwner, aPos, aSize, aPP, lVisible )

   DEFAULT oParent  TO SetAppWindow():setParent(), ;
           aPos     TO {148,98}, ;
           aSize    TO {334,182}, ;
           lVisible TO .F. , ;
           aPP      TO {}

   AAdd ( aPP, { XBP_PP_BGCLR, GRA_CLR_PALEGRAY } )
   AAdd ( aPP, { XBP_PP_COMPOUNDNAME, "8.Helv.normal" } )
   ::XbpDialog:init( oParent, oOwner, aPos, aSize, aPP, lVisible )
   ::XbpDialog:drawingArea:ColorBG := GRA_CLR_PALEGRAY
   ::XbpDialog:border := XBPDLG_SIZEBORDER
   ::XbpDialog:title := "New Form"
```

The default value for *oParent* (the parent window) is given by the expression *SetAppWindow():setParent()*, which is the desktop window in general. At this point, the same assumption is made as in functional code. If the desktop window should not be used as parent window, the parent must either be explicitly specified or the code of the *:init()* method must be adjusted (Note: if the form should be used as application window, an instance of the class is to be created in AppSys() and the return value of SetAppWindow() must be used as parent window. This function returns the desktop window when a program starts).

The major task of the *:init()* method is to initialize the super class (XbpDialog) and all contained Xbase Parts. The code for the initialization of Xbase Parts is very similar to functional code as described in the previous section. The only difference is that object references are assigned to instance variables instead of LOCAL variables. The following lines of code show a comparison of functional and object-oriented code which create the Group box in the sample form:

```
Functional code

   oXbp1 := XbpStatic():new( drawingArea, , ;
                             {12,48}, {300,96} )
   oXbp1:caption := "Customer"
   oXbp1:type := XBPSTATIC_TYPE_GROUPBOX


Object-oriented code

   ::GroupBox       := XbpStatic():new( ::drawingArea, , ;
                                        {12,48}, {300,96} )
   ::GroupBox:caption := "Customer"
   ::GroupBox:type := XBPSTATIC_TYPE_GROUPBOX
```

In functional style the LOCAL variables *oXbp1* and *drawingArea* are used, whereas the object-oriented code uses the instance variables *::GroupBox* and *::drawingArea* to store references to Xbase Parts. *::GroupBox* is declared in the *_CustomerForm* class and *::drawingArea* is an instance variable of the super class *XbpDialog* (Note: *::GroupBox* is an abbreviation for *self:GroupBox*, *::drawingArea* is the abbreviation for *self:drawingArea*).

The code that initializes Xbase Parts, which access database fields, is also similar in functional and object-oriented style:

```
Functional code

   oXbp := XbpSLE():new( oXbp1, , {96,48}, {192,24} )
   oXbp:bufferLength := 20
   oXbp:tabStop := .T.
   oXbp:dataLink := {|x| IIf( PCount()==0, ;
                         Trim( CUSTOMER->LASTNAME ), ;
                               CUSTOMER->LASTNAME := x ) }
   oXbp:create():setData()
   AAdd( aEditControls, oXbp )

Object-oriented code

   ::Lastname := XbpSLE():new( ::GroupBox, , {96,48}, {192,24} )
   ::Lastname:bufferLength := 20
   ::Lastname:tabStop := .T.
   ::Lastname:dataLink := {|x| IIf( PCount()==0, ;
                            Trim( (::CUSTOMER)->LASTNAME ) , ;
                                  (::CUSTOMER)->LASTNAME := x ) }
   AAdd( ::editControls, ::Lastname )
```

In object-oriented code, the XbpSLE object is referenced in the *::Lastname* instance variable. It has the same name, or symbol, as the database field that is edited by the XbpSLE object. When an Xbase Part accesses a field, the FormDesigner uses the field name as identifier for the corresponding instance variable by default.

The code for the *:dataLink* code block is also different in object-oriented code. Access to a field is programmed as *(Alias)->Fieldname* and the work area is determined by the contents of an instance variable. Functional code uses literal field names and aliases like *Alias->Fieldname*.

The final difference between functional and object-oriented code is the call to the *:create()* and *:setData()* methods. In functional code, both methods are called in the very same line of the program. In object-oriented code, these methods are called in two different classes: the method *:create()* is called in the *_CustomerDialog* class (Implementation level, _SAMPLE2.PRG) and *:setData()* is executed in the *CustomerDialog* class (Utilization level, SAMPLE2.PRG).

```
File: _SAMPLE2.PRG (Implementation level)

   CLASS _CustomerForm FROM XbpDialog

   METHOD _CustomerForm:create( ... )
      ::XbpDialog:create( ... )
      <...>
      ::Lastname:create()
      <...>
   RETURN self


File: SAMPLE2.PRG (Utilization level)

   CLASS CustomerForm FROM _CustomerForm

   METHOD CustomerForm:create( ... )

      * Execute method of the super class
      ::_CustomerForm:create( ... )

   // Path is abbreviated only in the documentation

      * Open databases and assign work area
      USE ..\SAMPLES\BASICS\XPPFD\CUSTOMER.DBF NEW EXCLUSIVE
      ::CUSTOMER := Select()

      * Transfer values to EditControls
      AEval ( ::EditControls, { | oXbp | oXbp:SetData() } )

      * Display the form
      ::show()

   RETURN self
```

The file _SAMPLE2.PRG contains the class declaration for the sample form plus the code required to create all Xbase Parts contained in the form. The file is the implementation level of the form. To utilize a form, it is necessary to open databases and to transfer values from the current record into the form. This is the task of a derived class. It is programmed in the file SAMPLE2.PRG and provides the utilization level of the form.

The source code for the utilization level may (and should) be modified when it is integrated into an application program. For instance, the file name of the database appears as literal name after the USE command and includes drive and path. This line definitely must be changed to reflect the situation in an application program. Whether or not databases are opened in the *:create()* method, or are already open when this method is called, must be decided by a programmer. The FormDesigner cannot solve such questions. It only makes sure that the generated code can be compiled in order to test the form in an isolated executable file.

To test the object-oriented code of a form, it is necessary to declare a Main procedure. This is optionally being done by the FormDesigner when the corresponding switch is set in the Settings window (check box: Main procedure for class code). In this case, the Main procedure is written to the file that contains the utilization level of a form. In the example, this is the file SAMPLE2.PRG. The Main procedure creates a form and processes events within a DO WHILE loop:

```
*******************************************************************
* Main procedure to test a form
*******************************************************************
PROCEDURE Main
   LOCAL nEvent, oXbp, mp1, mp2

   CustomerForm():New():Create()

   DO WHILE nEvent != xbeP_Close
      nEvent := AppEvent ( @mp1, @mp2, @oXbp )
      oXbp:HandleEvent ( nEvent, mp1, mp2 )
      IF nEvent == xbeP_Quit
         QUIT   // AppQuit()
      ENDIF
   ENDDO
RETURN

* Include program code for the implementation-level class of the form
#include "_SAMPLE2.PRG"

// EOF
//////
```

The main procedure, as generated by the FormDesigner, is only intended to test a form. Prior to the event loop, the class is instantiated without even assigning the object reference to a variable. The expression *CustomerForm():New():Create()* creates the form as defined in the class and stores the object reference in the child list of the application window (-> SetAppWindow():childList()[1]).

To create an executable file for testing a form, two PRG files must be compiled and linked (SAMPLE2.PRG and _SAMPLE2.PRG). This is accomplished by including the file containing the implementation level (_SAMPLE2.PRG) in the file with the utilization level (SAMPLE2.PRG). When the compiler compiles the file SAMPLE2.PRG, the preprocessor already has processed the *#include "_SAMPLE2.PRG"* directive and only one file needs to be compiled. This file contains the declaration for both classes, *CustomerForm* and *_CustomerForm.*

5.5. Using CLASS Code

To integrate the object-oriented code into an application program, it may (and should) be modified. It is recommended to make changes only in the file that contains the source code for the utilization level of a form. Then it is possible to change the form at a later point in time and have the FormDesigner generating new source code that reflects any changes. If a form is changed after source code is generated, the Form Designer generates only code for the implementation of a form. The file that contains the utilization level, or application specific code, remains unchanged.

An example of how object-oriented code can be changed is the file SAMPLE3.PRG which is located in the ..\SAMPLES\BASICS\XPPFD directory. The *CustomerForm* class from SAMPLE2.PRG is modified in such a way that the form can be used in a multi-user environment. Three methods are added to the class declaration and code for these methods is implemented.

```
CLASS CustomerForm FROM _CustomerForm
   EXPORTED:
   METHOD init
   METHOD create

   * Application specific methods (they are user-defined)
   METHOD skip
   METHOD setData
   METHOD getData
ENDCLASS
```

Before the record pointer is moved, all data changed in a form must be written to the database. In a multi-user environment this requires the record to be locked before it can be written to. This program logic is realized as application-specific code in the new methods

:skip(), *:setData()* and *:getData()* and is the only difference between the files SAMPLE2.PRG and SAMPLE3.PRG. The *:init()* method ensures that the new methods are executed:

```
METHOD CustomerForm:init( oParent, oOwner, aPos, aSize, aPP, lVisible )

   * Execute method of the super class
   ::_CustomerForm:init( oParent, oOwner, aPos, aSize, aPP, lVisible )

   * Change code blocks for pushbuttons
   ::ButtonOK:activate       := {|| ::getData(),;
                                    PostAppEvent( xbeP_Close ) }
   ::ButtonPrevious:activate := {|| ::skip(-1) }
   ::ButtonNext:activate     := {|| ::skip( 1) }
RETURN self
```

The pushbuttons contained in the form are referenced in instance variables which are declared in the *_CustomerForm* class (_SAMPLE2.PRG). The *:activate* code blocks are changed, so the new methods will be executed when a user clicks a pushbutton. The implementation of the new methods requires only a few lines of code:

```
METHOD CustomerForm:skip( nSkip )
   IF ::getData()                      // Write record
      (::CUSTOMER)->(DbSkip( nSkip ))  // Change record pointer
      ::setData()                      // Read record
   ENDIF
RETURN self


METHOD CustomerForm:setData
   AEval( ::editControls, {|o| o:setData() } )
RETURN self


METHOD CustomerForm:getData
   LOCAL lOk := ( AScan( ::editControls, {|o| o:changed } ) == 0 )

   IF ! lOk                            // Data is changed
      lOk := (::CUSTOMER)->(DbRLock()) // Lock record
      IF lOk                           // Record is locked
         AEval( ::editControls, {|o| o:getData() } )
         (::CUSTOMER)->(DbUnlock())    // Unlock record
      ELSE
         MsgBox( "Record is currently locked" )
      ENDIF
   ENDIF
RETURN lOk
```

The *:skip()* method navigates the record pointer in this example only if changed data in the form is written to the database. This, in turn, depends on a successful record lock which is set in the *:getData()* method. If data is unchanged, no locking occurs and the record pointer is always moved. When the record pointer is changed, the *:setData()* method transfers data from the database into the form.

6. The Xbase++ compiler - XPP.EXE

The Xbase++ compiler can be started by entering XPP on the command line. The general syntax of the call to the compiler is as follows:

```
XPP [<Options>] <PRG-file>     //  or
XPP <PRG-File> [<Options>]
```

<PRG-File> indicates the file name of the source code file the to compile. If the file name is specified without an extension, .PRG is used as the file extension. <Options> set switches to control the compile process. A compiler switch always starts with a slash ("/") or a hyphen ("-") (see below).

The program XPP.EXE contains two components: the preprocessor and the actual compiler. XPP.EXE translates a source code file in ASCII format to an OBJ file in binary format. The translation of the source code occurs in two steps. First the preprocessor translates the program code within the PRG file into modified code which the compiler "understands". Then the compiler creates an OBJ file which the linker is able to link into an executable program.

The result of the preprocessor can be output as a separate file with the extension PPO (PPO stands for preprocessed output) using the compiler switch /p. A PPO file is an ASCII file containing the code that the compiler actually compiles. The preprocessor prepares program code for the compiler and the compiler converts the prepared code into an OBJ file.

6.1. Compiler switches

The compilation of ASCII program code (either a PRG file or a PPO file) into binary program code can be controlled in specific ways using compiler switches. These switches must be specified along with the file name of the source code file when XPP.EXE is called. The following switches (<Options>) are valid for the compiler:

/?	Display information about all compiler switches. The /? switch only provides information and can not be combined with other switches.
/a	Automatic MEMVAR declaration When this switch is used, all variables specified with PRIVATE, PUBLIC or PARAMETERS are explicitly declared as MEMVAR.
/b	Insert debug information When /b is used, debug information is written into the resulting OBJ file. This includes row numbers, names of lexical variables,

the source code file name and other information.

If the /b switch is not included, the debugger can not display information about the program source code at runtime of the resulting Xbase^{++} program. The debug information increases the size of both the OBJ file and the executable EXE file. For this reason, the final version of a program should be compiled and linked without /b.

/coff Creates object files in Common Object File Format (COFF)

The Xbase^{++} compiler creates COFF object files if the switch /coff is set. On Windows platforms, this switch is activated by default. To create OMF object files, the /omf switch must be used.

/com Use compatibility mode

In the compatibility mode, all identifiers (names) for functions, procedures, methods and variables are considered significant by the compiler only up to a maximum of 10 characters. The /com switch activates this mode. It should only be set when compiling Clipper programs with function names that are actually longer than 10 characters but abbreviated using only the first 10 characters. The side effect of the /com switch is that all Xbase^{++} functions with identifiers more than 10 characters long can no longer be called.

/d<id>[=<val>] #define <id>

The /d switch specifies the #define constant <id> to the compiler on the command line. The #define constant is valid within the source code file. Optionally, the constant can be assigned the value <val>.

/dll[:DYNAMIC] Create OBJ file for a DLL file

The /dll switch directs the compiler to include additional information necessary for the creation of a DLL file in the OBJ file. All OBJ files which are to be linked into a DLL file must be compiled using /dll. If the linker will create an EXE file, the switch must not be used.

Important: If a DLL file is to be loaded and released during runtime of an Xbase^{++} application using the functions DllLoad() and DllUnload(), the additional keyword :DYNAMIC must be specified.

/err:<count>	Terminate compiling after <count> errors By default, the compiler terminates the compile process as soon as it registers 20 program errors. This value can be changed using the /err: switch. <count> specifies the maximum error count before compiler termination. In some cases the number of 20 errors is reached very quickly. When the compiler detects a syntax error it tries to find the next entry point in the source code to continue the compile process with the remaining code. If no entry point is found after an initial error the compiler reports additional syntax errors even if the code is correct.
/es	Creates a operating system error code for warnings When the switch /es is set the compiler creates an OS error code already for warnings and not when a syntax error is detected.
/ga	Converts literal strings to ANSI When the /ga switch is set, all literal character strings in the PRG source code are converted to ANSI before the compiler creates the OBJ file.
/go	Converts literal strings to OEM When the /go switch is set, all literal character strings in the PRG source code are converted to OEM before the compiler creates the OBJ file.
/i<path>	Search directory for #include files The /i switch specifies an additional search directory *<path>* for the compiler to use when locating #include files. Normally, the compiler only searches for these files in the current directory and those directories specified by the INCLUDE environment variable.
/l	Insert no line numbers If the /l switch is used, no line numbers are included in the OBJ file. This decreases the size of the executable EXE file, but has the disadvantage that it disables ability of the function ProcLine() to return line numbers. If /l is used and a runtime error occurs, it will not be possible to determine which line in the source code caused the error.

/link[:"options"]	Creates an EXE from the object file The /link switch causes the compiler to start the linker when the OBJ file is successfully created. In this way it is possible to create an EXE file from a single PRG file by invoking the compiler only. Options for the linker can be specified with the /link switch as well. They must be enclosed in double quotes and are passed on to the linker.
/m	Ignore SET PROCEDURE TO (ProcRequest) The /m switch prevents automatic insertion of additional source code files (PRG files) which are specified using the command SET PROCEDURE TO. Since SET PROCEDURE TO is purely a compatibility command, the /m switch does not generally need to be used.
/n	No implicit start procedure (MAIN) If program code is found in a source code file outside a FUNCTION, PROCEDURE or METHOD declaration, this program code is implicitly condensed in the procedure MAIN(). This default behavior can be suppressed by the /n switch. The compiler then creates error messages if it finds program code outside of a FUNCTION, PROCEDURE or METHOD declaration.
/nod	No default library in OBJ file If the /nod (no default library) switch is used, the name of the file XPPRT1.LIB is not implicitly included in the OBJ file. The linker then only uses the LIB files specified with the /r switch.
/o<name>	Rename OBJ file Normally, the compiler creates an OBJ file which has the same file name as the PRG file it is created from. The /o switch is used to rename the resulting OBJ file to *<name>*. If a different path is used in *<name>*, specifying the path name is sufficient. The name of the path must be terminated using a backslash \. The OBJ file is then created in the specified path with the same file name as the PRG file.

/omf

Creates object files in Object Module Format (OMF)

The Xbase⁺⁺ compiler creates OMF object files if the switch /omf is set. On the OS/2 platform, this switch is activated by default. To create COFF object files, the /coff switch must be used.

/p

Create preprocessor output (PPO file)

If the /p swtich is included, the compiler creates a PPO file containing the preprocessor output in addition to creating the OBJ file. The file has the same name as the source code file, but has PPO instead of PRG as the file extension. A PPO file can be used to determine whether user defined commands are correctly translated by the preprocessor.

/pre[:<min>]

Load DLL files of the compiler

The /pre (preload) switch indicates that all DLL files used by the compiler are loaded into main memory and remain available in memory after the compile terminates. A time interval in minutes can optionally be specified using :<min>. When no compiling process has taken place after :<min>, the DLL files are automatically removed from memory. The default value for :<min> is 10 minutes. This means that if no time interval is specified, the DLL files are removed from memory after 10 minutes.

/pre is used to avoid the time consuming loading and removing of the DLL files in the path \XPPW32\BIN and accelerates repeated compiling.

/q

No screen display during the compiling (quiet)

The /q switch suppresses the display of line numbers during compilation. This allows the compiler to work faster.

/r<libname>

Specify LIB file for linker

The /r switch is used to specify an additional library <libname> for the linker to search in order to resolve external references when linking the OBJ file. By default, the linker searches the file XPPRT1.LIB and it does not need to be explicitly specified unless the /nod switch is used.

/s	Only test syntax If the /s switch is used, the compiler merely performs a syntax check of the source code file and does not create an OBJ file.
/u[<name>]	Use user-defined STD.CH The /u switch defines the #include file <name> as a replacement for the file STD.CH which is included by default in all source code files during compilation. A programmer who only uses commands to a very limited extent can copy these commands from the file STD.CH to another file and specify this file as the default using /u. This saves time when the preprocessor reads the #include file. If <name> is not specified, no default #include file is used.
/v	Treat undeclared variables as MEMVAR By default the compiler treats undeclared variables which are not preceded by alias names as field variables. The /v switch causes the compiler to treat these undeclared variables as dynamic memory variables. These variables then receive the alias name MEMVAR->.
/w	Display warnings The /w switch causes the compiler to output warnings on the screen when it detects undeclared variables which are not preceded by alias names.
/wi	Display warnings for non-initialized variables The /wi switch produces warnings for all variables which are not initialized and appear in expressions where the variables must have a value.
/wl	Display warnings for non-lexical variables The /wl switch produces warnings for all variables which are not declared as LOCAL or STATIC or included in the formal parameter list of functions, procedures and methods.
/wu	Display warnings for unused lexical variables The /wu switch produces warnings for all lexical variables which

	are declared and not used.
/z	Turn off logical short-cut optimization The /z switch turns off the default "short-cut" optimization for the logical operators .AND. and .OR.. Without optimization, all expressions which are combined using logical operators are evaluated. With optimization, combined expressions are evaluated only until the result of the overall expression is clearly determined.

6.2. Examples for the Xbase++ compiler

The following example performs a syntax check for the file MAIN.PRG and writes the preprocessor output into the file MAIN.PPO without displaying line numbers.

```
XPP MAIN /s /p /q
```

The next example creates the file TEST.OBJ including debug information. The OBJ file is written to the path \XPP\OBJ\. No implicit MAIN() procedure is created and warnings are displayed on the screen during compilation:

```
XPP TEST /b /n /w /o\XPP\OBJ\
```

Constants defined by the compiler

By default, the Xbase++ compiler defines two constants: one to identify the compiler and the other one to identify the operating system. This allows program code specific for a compiler or an operating system to be excluded from compilation, using the directives *#ifdef, #ifndef, #else* and *#endif.* The *#define* constants are:

```
#define __XPP__       // Xbase++
#define __OS2__       // OS/2 version
#define __WIN32__     // Windows 32 bit version
```

For example, if the same PRG file is to be compiled by Xbase++ and Clipper, there may be situations where both compilers behave differently. Such conflicts can be resolved as follows:

```
#ifdef __XPP__
    <Xbase++ Code>
#else
    <Clipper Code>
#endif
```

In this way, program code for different compilers can be maintained in the very same PRG file.

6.3. Version information

The utility program XPPLOAD.EXE can explicitly load all compiler DLL files into main memory without actually starting the compiler. It displays version information when it is invoked on the command line as follows:

```
XPPLOAD version
```

7. The Xbase++ Debugger - XPPDBG.EXE

This chapter describes the debugger provided with Xbase++. The debugger is a powerful tool for program development and helps the programmer find errors in applications running in the VIO, hybrid, or GUI mode. The debugger itself runs in an XbpCrt window and is primarily written in Xbase++.

7.1. Basics of the debugger

The Xbase++ debugger is started by entering XPPDBG on the command line and runs as an independent process. If the filename of an Xbase++ application is specified as a command line argument, this application is started as another process and debug information about the Xbase++ application can be displayed. For debug information to be available, all of the PRG files of the Xbase++ application must have been compiled using the compiler option /b and the resulting OBJ files linked using the linker option /DE. Otherwise no debug information is available.

The debugger primarily displays debug information about the source code of the application being executed and controlled by the debugger. Any memory variable can be displayed and its contents edited at runtime using the debugger. This is possible whether the variable is declared LOCAL, STATIC, PRIVATE, or PUBLIC. The debugger can also display the callstack, that lists all default user-defined functions, procedures, and methods.

The debugger can interrupt the running application and control the application via breakpoints or using single step mode (this means the source code of the application is processed line by line). Breakpoints automatically interrupt the application and reactivate the debugger. This allows the application to be run step by step to limit and localize the location of an error. The debugger also includes a command line where valid Xbase++ expressions can be executed within the application.

Navigating within the debugger is done using the mouse or short-cut keys. The most important short-cut keys are F8, F5 and the key combination Ctrl+S. F8 executes the next source code line of the application when in single step mode. The F5 key ends the single step mode of the debugger and returns control back to the application until the debugger is reactivated using Ctrl+S (stop application) or a breakpoint is reached. Most other aspects of the debugger are menu driven. The current activity of the debugger is shown by a colored area within the menu bar. Red means the debugger is inactive and the application is running. Green means the debugger is active and controls the application.

Important note: When the application is active, the application can only be interrupted using the key combination Ctrl+S when program code is being executed in the application. If the application is in a wait state, it is not executing code. This happens during WAIT, Inkey(0), and AppEvent(). As long as the application is waiting for keyboard entry or an event, the debugger can not intervene in the application since no code is being executed in the application. In this case, the window of the application must be brought into the foreground and a key pressed if necessary so that the application ends the wait state. Only then can the debugger regain control.

7.2. The menu system of the debugger

The main menu of the debugger is always visible as the top line of the debugger window. It contains the following menu items:

File	Activates the file menu. This allows navigation within the code window, selects application source code files for display in the code window and terminates the debugger.
Run	Activates the run menu. Run menu options control how the application is run, restart the application and can select another Xbase^{++} application to debug.
Monitor	Activates the monitor menu. This opens or closes the monitor window for displaying memory variables and the callstack window for displaying the active functions and procedures.
Commands	Activates the command menu. Command menu options control the command window of the debugger and allow breakpoints to be set or deleted.
Options	Activates the options menu. This menu configures the debugger and allows the current configuration to be saved.
Help	Displays a help window.

7.2.1. File menu

The file menu contains options that specify the source code file displayed in the debugger's code window and what section of the source code is visible.

Open module This option opens a file dialog that lists all the PRG files containing source code. A selection is made in the window using the Return key or by double clicking the highlighted item. The selected PRG file is displayed in the code window of the debugger and breakpoints can be set in this file using the F9 key or Return.

A selection window can also be opened containing a list of all

	functions, procedures, and methods contained in the PRG file. This selection window is opened by clicking the right mouse button or using the key combination Ctrl+Return after a PRG file is selected.
Goto Line	This option opens an input window that allows specification of a line number for the debugger to display in the code window.
Goto Function	This option displays the source code of any function or procedure in the code window. If the source code for the function is not in the PRG file that is currently displayed in the code window, the debugger opens the appropriate PRG file. If several functions with the same name exist (STATIC FUNCTION) the debugger displays the source code of the first function it finds with the specified name.
Where	This option displays the current program line being executed in the code window. The appropriate PRG file is loaded if it is not already visible. This option performs the same functions as "Open module" and "Goto ..." to reposition the source code on the current line after the code window is scrolled or another window is opened.
Exit	This option terminates the debugger.

7.2.2. Run menu

This menu contains options to control the Xbase^{++} application. This includes selecting and restarting the application.

Restart	This option terminates and then restarts the current Xbase^{++} application.
Startup	The option terminates the current application and opens a window where an application file name can be entered. This allows a different Xbase^{++} applications to be debugged without terminating and restarting the debugger.
Step	The short-cut key for this option is F8. Either selecting the menu option or pressing F8 executes the next program line in the application source code.
Trace	The short-cut key for this option is F10. Pressing F10 and selecting this menu option both cause all functions and procedures called by the current program line to be completely executed before the debugger pauses program execution on the next line. This skips stepping through the source code of the calling function or procedure.

Go	F5 is the short-cut key for this option. This returns control to the application and turns the debugger off. The debugger can only regain control by being reactivated with the key combination Ctrl+S or if a breakpoint occurs. If Ctrl+S is pressed, the debugger window receives input focus, meaning it must be in the foreground. If the debugger does not display a "green light" after Ctrl+S, the application is in a wait state condition and does not execute any code. In this case the application must receive an event so that it leaves the wait state. The easiest way to accomplish this is to click on the application window with the mouse and if necessary press a key.
Goto Cursor	F7 is the short-cut key for this option. The debugger lets the program code of the application run to the current line in the code window and then interrupts program execution. This menu option and the F7 key set the current line in the code window as a temporary breakpoint.
Step Back	The short-cut key for this option is F4. This option effectively uses the line after the current procedure or function as a break point. When pressed, the current procedure or function runs to completion. The debugger interrupts the program as soon as the current function is terminated. It then positions the code in the code window on the next line of the source code file that called the function or procedure that was active when F4 was pressed.

7.2.3. Monitor menu

This menu includes options to open windows for viewing memory variables and/or the callstack. The information in the windows is automatically updated while the application is running. When the monitor window is active, a variable can be selected with a double click and then edited. The corresponding PRG file is automatically loaded into the code window when any one of the functions or procedures in the callstack window is double clicked with the left mouse button.

Local	Turn the display of variables of the LOCAL storage class on or off.
Static	Turn the display of variables of the STATIC storage class on or off.
Private	Turn the display of variables of the PRIVATE storage class on or off.
Public	Turn the display of variables of the PUBLIC storage class on or off.
Callstack	Turn the display of the callstack on or off.

7.2.4. Commands menu

The command window is opened or closed using options in this menu. This window displays the result of all expressions entered on the command line of the debugger. It also records the expression entered and their result. The up arrow key can be used to retrieve a previously entered expression which can be executed again by pressing Return. In addition to managing the command window, the menu also manages breakpoints. A breakpoint designates a line in the source code of the application where the program is automatically interrupted. If a breakpoint is reached while the application is running, the debugger is reactivated and regains control.

Toggle Breakpoint	F9 is the short-cut key for this option. Alternatively, the Return key can be pressed. This option defines the current line in the application source code as a breakpoint or clears a previously defined breakpoint on this line. Whether or not a line is defined as a breakpoint is indicated using the character » , which appears at the beginning of the source code line in front of the line number.
Delete All Breakpoints	This option deletes all of the defined breakpoints in all source code files.
List Breakpoints	This option lists all defined breakpoints in all source code files.
Open Command Window	This option opens the command window. Expressions can then be entered on the command line of the debugger. When the command window is already opened, this menu option is replaced with "Close Command Window".
Clear Command Window	This option deletes all the lines previously recorded in the command window.

7.2.5. Options menu

This menu allows several aspects of the debugger to be configured or the current window configuration to be stored. Configuration data is saved in a file with the extension .@@@ and the same file name as the EXE file. If this file exists in the current directory, it is read by the debugger and used for configuration.

Debug Program Startup	If this option is selected, the debugger also displays the program code executed before the call to the Main

	procedure. This displays the functions AppSys(), DbeSys() and any other INIT PROCEDURE.
Check Time Stamps	This option can be turned on or off. If it is activated, the debugger compares the time stamp of the source code files of the application with the time stamp on the executable files and gives a warning if the executable files (DLL and EXE files) are older than one of the source code files.
Set Tab Width	This option specifies the number of blank spaces that are used to replace tab characters when displaying source code in the code window of the debugger.
Save Restart Info	This option saves the current debugger configuration and all current breakpoints. The file created has the same file name as the application and the extension ".@@@". If XPPDBG **-n** <EXE-file> is specified on the command line, the debugger ignores the saved information and does not load the @@@ file.

7.2.6. Help menu

This menu contains options that display help information for the debugger.

Short Help	This option opens a small help window that displays the short-cut keys for controlling the debugger.

7.3. Working with the debugger

The debugger's purpose is to help the programmer find and resolve program errors. This section discusses the capabilities offered by the debugger and offers advice on how to locate errors. Program errors leading to program termination must be differentiated from those that can not be identified by the runtime system. Errors that cause program termination are relatively easy to locate, since the default error handling routine displays information about the type and location of the error on the screen. The most important information for locating a runtime error is the name of the routine and the source code line number in the PRG file where the error occurred. If the reason for the error is not obvious, a breakpoint can be set on this line in the debugger and the conditions leading to the runtime error examined with the debugger.

7.3.1. Setting breakpoints

The debugger is started by entering XPPDBG <EXE file> on the command line. It loads the executable file <EXE file> and stops the Xbase^{++} application on the first executable program line. An implicit breakpoint is set on this line. If a runtime error occurs, the debugger identifies the name of the routine where the error occurred along with the line number in the PRG file. To set a breakpoint at this line, the option "Open Module" is selected in the file menu. This displays a selection window containing the names of all the PRG files. After the highlight is positioned on the name of the file where the routine causing the error is found, clicking the right mouse button on the highlighted file (or the key combination Ctrl+Return) will open a second window listing all the routines contained in the PRG file. The routine that generated the error can be selected from this list. Double clicking with the left mouse button or pressing the Return key opens the corresponding PRG file and positions the code window to the selected routine.

Within the code window the highlight should be positioned on the program line where the error occurred. A breakpoint is set by double clicking the left mouse button on the highlighted line or by pressing F9 or the Return key.

A debug session generally starts by setting breakpoints. Any number of breakpoints can be set. The option "Save Restart Info" can be selected in the options menu to save the breakpoints from one execution of the application to the next. The line numbers of the breakpoints are then saved in a file with the same file name as the application and the extension ".@@@".

After breakpoints are set, the application is run by pressing the F5 key. The colored area in the menu bar of the debugger changes from green to red. This indicates that the application, and not the debugger, has input focus. The debugger interrupts the application and regains control as soon as a breakpoint is reached. The colored highlight then changes from red back to green.

7.3.2. Inspecting errors

As soon as the debugger has interrupted the application at a breakpoint, the current status of the application can be more closely examined. The monitor window is opened from the monitor menu to display the contents of all visible variables. Variables are organized based on their storage class (LOCAL, STATIC, PRIVATE and PUBLIC). If a variable has an incorrect value, it can be changed using the debugger. To do this, the monitor window is selected by clicking the left mouse button or using the Tab key. Within the monitor window the highlight must be positioned on the corresponding variable. An edit window is then opened by clicking the left mouse button or pressing Return. The value of the variable can be edited in this window. The edit window is closed by selecting OK.

Variables with the data type character, numeric, date or logical can be displayed in an entry field where they can be edited. The display uses Xbase^{++} syntax. Code blocks can not be displayed but can be modified. When changing a value, pay careful attention to the syntax.

For example, delimiters must be present for a character string when a value of the character data type is edited. If the variable is a code block, the characters {|| and } must be included. The characters entered are compiled using the macro operator and the result is assigned to the variable.

If the variable to be edited is an array, the contents of the array elements can be viewed using the View pushbutton. The maximum number of elements displayed is 64. Elements in an array beyond this 64 element limit can be examined by entering the numeric index for the first array element to display in the entry field "Start".

The edit window displays only the essential information about an object, such as the name of the class to which the object belongs. The contents of the instance variables of an object are displayed by pressing the View pushbutton. Only the instance variables declared in the class are displayed. If the class is derived from other classes, the class names of the superclasses are also displayed. Selecting a superclass displays an edit window containing the instance variables for this class.

Only the instance variables of objects can be examined in the edit window of the debugger. In many cases the return value of a method needs to be checked. For example, the question might be "What is the class name of the parent of this Xbase Part?" The parent can only be determined using the method *:setParent()*: it does not appear in the list of instance variables. Problems such as this are solved using the command window of the debugger. To view the command window the option "Open Command Window" is selected to open the command window. Below this window the command line of the debugger is displayed and any expressions can be entered. After pressing the Return key the expression is executed and the result of the expression is displayed in the command window.

The parent of a particular Xbase Part can be determined on the command line of the debugger in two ways. If the variable oXbp contains an Xbase Part whose parent information is needed, either of the following two expressions can be used on the command line of the debugger to determine the parent class:

```
oXbp:setParent():className()

dummy := oXbp:setParent()
```

The first expression displays the class name of the parent in the command window. The second expression creates a PRIVATE variable referencing the parent. Selecting the option "Private" for the monitor window displays variables of the PRIVATE storage class. The parent of oXbp is displayed as the variable "dummy". This variable can be examined with the debugger in the manner described earlier.

8. The Xbase++ ProjectBuilder - PBUILD.EXE

The Xbase++ ProjectBuilder is a tool for managing entire software projects. A project consists of at least one EXE file but can be comprised of several EXE and/or DLL files as well. The description of a project is contained in a project file. This is an ASCII file with the extension XPJ (Xbase++ ProJect). It lists all necessary data for building a project. For example, names of source files, information for compiler and linker or which executable file must be created from which source files.

8.1. Creating a project file (XPJ file)

The easiest way to create a project file is by using an ASCII file which lists the names of all PRG sources which are part of a project. Such a file can be created with the DIR command:

```
DIR /b *.prg > project.txt

PBUILD @project.txt
```

The output of the command DIR is directed into the file PROJECT.TXT. This file can easily be modified using a simple text editor to remove names of PRG files which are located in the current directory but are not part of the project, for example. The ProjectBuilder (PBUILD.EXE) reads such a file and creates from it a template for a project file with the extension XPJ (this stands for Xbase++ ProJect).

For small projects, the basic structure of a project file can also be created quickly with an editor. The following example shows a project file for the program CUSTOMER.EXE which needs only four PRG files:

```
[PROJECT]
   DEBUG = yes        // Project-wide definitions
   GUI   = no
   CUSTOMER.XPJ       // The root of the project

[CUSTOMER.XPJ]        // List all EXE and DLL
   CUSTOMER.EXE       // files of the project here

[CUSTOMER.EXE]        // List all sources for each
   CUSTOMER.PRG       // EXE and/or DLL file separately
   GETCUST.PRG
   PRINT.PRG
   VIEWCUST.PRG
```

The example shows the major characteristic of an XPJ file: it is divided into different sections, each of which begins with a symbolic name enclosed in square brackets. The first section must always be named [PROJECT]. It is the entry point for the ProjectBuilder and lists definitions valid for the entire project. In this example, a program containing debug information is to be created as text mode application (lines 2 and 3). At the end of the definitions list (line 4), the name of the next section which is to be analyzed by the ProjectBuilder appears. This section is the root of the project and lists all executable files which are part of the project, or which must be distributed later to customers, respectively. This includes both EXE and DLL files. Each name of an executable file is used again as the name for another section which lists the names of the source files the executable file is created from.

As a result, the structure of an XPJ file represents the dependencies which exist between source and target files of a project. For example, the beforementioned file CUSTOMER.EXE is a target file which depends on four PRG files, or sources, respectively. Whenever a source is changed, the target must be updated. The ProjectBuilder analyzes the dependencies between source and target using a project file and updates an entire project to reflect the latest changes.

The example above shows only the basic structure of an XPJ file as it can be easily created with an editor. However, many more dependencies can exist between different files of the same project. These include, for example, dependencies between CH->PRG, PRG->OBJ, OBJ->EXE or OBJ->DLL and DEF->LIB files. The ProjectBuilder is able to detect all of these numerous dependencies automatically once the basic structure of a project file is created. For this, PBUILD.EXE is started with the /g switch (g for generate):

```
PBUILD customer.xpj /g
```

The switch /g causes PBUILD.EXE to analyze all dependencies that exist within a project. The ProjectBuilder then expands a (rudimentary) project file and adds all missing information to the XPJ file. This frees a programmer from quite a lot of typing, especially for large projects. For instance, PBUILD.EXE expands the previous example XPJ file by 20 lines. As a result, the project file grows from 11 to 31 lines:

```
01:  [PROJECT]
02:      COMPILE       = xpp       // Missing compiler and linker
03:      COMPILE_FLAGS = /q        // information is added
04:      DEBUG         = yes
05:      GUI           = no
06:      LINKER        = alink
07:      LINK_FLAGS    =
08:      RC_COMPILE    = arc
09:      RC_FLAGS      =
10:      CUSTOMER.XPJ
11:
12:  [CUSTOMER.XPJ]
13:      CUSTOMER.EXE
14:
```

```
15:   [CUSTOMER.EXE]                // Automatically created
16:   // $START-AUTODEPEND          // dependencies
17:       COLLAT.CH
18:       GET.CH
19:       MEMVAR.CH
20:       PROMPT.CH
21:       SET.CH
22:       STD.CH
23:       CUSTOMER.OBJ
24:       GETCUST.OBJ
25:       PRINT.OBJ
26:       VIEWCUST.OBJ
27:   // $STOP-AUTODEPEND
28:       CUSTOMER.PRG
29:       GETCUST.PRG
30:       PRINT.PRG
31:       VIEWCUST.PRG
```

Changes appear in the [PROJECT] section, where all missing information for compiling and linking is added. Furthermore, the section [CUSTOMER.EXE] now contains a list of files for which the ProjectBuilder has detected dependencies. This automatically created list is enclosed by the markers // $START-AUTODEPEND and // $STOP-AUTODEPEND in lines 16 and 27. Both markers indicate an area in the project file which may be changed automatically when PBUILD.EXE is called subsequently together with the /g switch. Therefore, a project file should not be altered between these markers as changes in this area can become lost.

8.2. Creating a project

When a project file including all dependencies is generated, a project is created, or updated, respectively, by invoking PBUILD.EXE and specifying the appropriate project file, if necessary. By default, the ProjectBuilder searches for the file PROJECT.XPJ in the current directory. Therefore, each of the following possibilities create a project:

```
1)  PBUILD

2)  PBUILD customer

3)  PBUILD customer.prj
```

In the first call, the ProjectBuilder creates the project which is described in the PROJECT.XPJ file. The second call requires that the file CUSTOMER.XPJ be available in the current directory (XPJ is the default extension for project files), while in the third call, a complete filename specifies the project file. After being invoked, the ProjectBuilder analyzes the dependencies between source and target files. If a source file has been changed since the last call to PBUILD.EXE, i.e. if source files are newer than target files, the ProjectBuilder creates the corresponding targets again (EXE or DLL files). Since only changed sources are treated by the ProjectBuilder, the time to update complex projects is therefore reduced to a great extent.

8.3. The [PROJECT] section in an XPJ file

All information that may be listed in the [PROJECT] section is described below. Each project file must begin with the [PROJECT] section which contains definitions valid for the entire project (project-wide definitions):

COMPILE=	This indicates the name of the Xbase^{++} compiler. It is always XPP.
COMPILE_FLAGS=	All compiler switches to be set for compilation are defined here. Note that separate definitions OBJ_DIR= and DEBUG= exist for the switches /o and /b.
DEBUG=	This definition can be set to YES (debug version) or NO (non debug version). Executable files are created accordingly with or without debug information. An executable file must be created with DEBUG=YES so that it can be monitored with the debugger.
GUI=	If a program is to run as text mode application, GUI=NO must be set. Whenever a program performs graphic output, be it with Gra..() functions or by using Xbase Parts, GUI=YES must be defined.
LINKER=	This definition indicates the name of the linker that is used to create EXE or DLL files from OBJ files. It is the linker that is shipped with the operating system-specific Xbase^{++} version.
LINK_FLAGS=	All linker flags which are not covered by GUI= and DEBUG= are listed in this definition. However, if the flags /PM and /DE are used here as well, they override the corresponding definitions GUI= and DEBUG=.
OBJ_DIR=	Optionally, the directory for OBJ files can be defined. The compiler creates OBJ files in this directory, and the linker searches it for these files.
RC_COMPILE=	This definition contains the name of the resource compiler as it is shipped with Xbase^{++}. Normally, additional resources are used for GUI applications only.
RC_FLAGS=	The flags for the resource compiler are set with this definition.

<SECTION>	All entries in the [PROJECT] section defined without equal signs are used as a reference to subsequent user-defined sections which are to be analyzed by the ProjectBuilder. Normally, only one additional section (the root section) is referenced. There must be at least one user-defined section.

Note: Definitions listed in the [PROJECT] section are valid for the entire project. However, they may appear in user-defined sections as well. In this case, they are valid for one section only. If definitions are passed to PBUILD.EXE on the command line using the /d switch, all definitions in a project file with the same name are ignored.

8.4. User-defined sections in an XPJ file

After the [PROJECT] section, user-defined sections are listed in a project file. The name of the first user-defined section (the root section) must be indicated in the [PROJECT] section, as shown in the following example:

```
[PROJECT]
   DEBUG = YES          // project-wide definition

   ROOT                 // first user-defined section

[ROOT]
   file1.EXE
   file2.DLL

[file1.EXE]
   <PRG files for the EXE file>

[file2.DLL]
   DEBUG = NO           // section-wide definition

   <PRG files for the DLL file>
```

The first user-defined section lists all executable files (targets) which are to be created by the ProjectBuilder. For each target, a user-defined section which has the same name exists. It lists the PRG files (sources) for that target. Sources are normally PRG files only, but can be ARC files as well if needed by a GUI application. ARC files are compiled with the resource compiler ARC.EXE and linked to the executable file.

When a target file is to be created as a dynamic library (DLL file), the corresponding file name must include the DLL extension. The ProjectBuilder automatically creates all necessary files for creating a DLL. This includes the export definition file (DEF file) and the import library (LIB file). The latter must be linked to an EXE that uses functions contained in a DLL.

When a project file contains multiple user-defined sections, it is sufficient to write only the source files to each section (PRG files and ARC files, if necessary). The project file can then be expanded to describe all dependecies between sources and targets by invoking PBUILD.EXE with the /g switch.

8.5. Options for PBUILD.EXE

The ProjectBuilder is started from the command line by entering PBUILD. If no project file is specified, the ProjectBuilder searches for the file PROJECT.XPJ and creates the project described in that file. The general syntax for the call is:

```
PBUILD @<file>
```

or

```
PBUILD [<XPJ file>] [<options>]
```

@<file>	The character @ indicates a file which lists all sources which are part of a project. *<file>* is an ASCII file that contains the name of one source file in each line. It can be created very easily by entering DIR /b *.PRG > PROJECT.LST on the command line, for example. From this file, the ProjectBuilder creates a project file with the same file name but a different file extension (XPJ). The name *<file>* is also used for the executable file to be created.
<XPJ file>	When a project is not described in the file PROJECT.XPJ, the name of the project file must be specified on the command line. The default extension is XPJ.
/? or /h	Display information about options of PBUILD.EXE. The /? switch only provides information and cannot be combined with other switches.
/a	The switch /a causes the ProjectBuilder to perform a complete compile and link cycle for all sources of a project. This rebuilds an entire project regardless of whether or not source files have been changed since the last update of the project.
/d<id>[=<val>]	The definition *<id>* in a project file can be set to the value *<val>* using the /d switch on the command line. For example, entering PBUILD /dDEBUG=NO builds a project without debug information even if DEBUG=YES is specified in the [PROJECT] section. The switch /d, therefore, temporarily changes definitions without changing the project file.

/g[<name>]	Specifying the switch /g causes the ProjectBuilder to analyze a project for dependencies. It then generates a list of all files the target file depends on. This option must be used to expand the basic structure of a project file or when dependencies are changed. The basic structure of an XPJ file covers only PRG files and targets (EXE or DLL). A complete XPJ file, however, lists files with the extension CH, OBJ, ARC, DEF and LIB (DEF and LIB files are used for creating DLL files). Optionally, a new project file can be created by specifying <*name*>. If the file name is omitted, an existing project file is overwritten.
/l	When DLL files are part of a project, the ProjectBuilder automatically creates corresponding DEF files (Export definition files) and LIB files (import libraries). Normally this is not necessary when a PRG file is changed but only when the number, sequence or identifiers of exported functions change in the DLL. For example, if a function in a PRG file is changed by inserting a few lines of code, the PRG file must be compiled and the DLL must be linked. However, the export definitions and the import library do not need to be rebuild in this case. Therefore, the /l switch suppresses automatic creation of DEF and LIB files.
/n	With /n, the ProjectBuilder only displays all necessary steps to update a project without performing the corresponding actions (compiler and linker are not invoked).
/v	The switch /v activates the verbose mode of the ProjectBuilder.

9. The Alaska Linker - ALINK.EXE

The linker ALINK creates a single file containing executable 32 bit code from OBJ and LIB (libraries) files. OBJ files are the program modules of an application created by the compiler and LIB files contain collections of several OBJ files. The output file of the linker can be either an EXE or a DLL file. All files to be linked by ALINK must comply with the Common Object File Format (COFF).

Note: If an application is created by the ProjectBuilder PBUILD.EXE it is not necessary to start the linker explicitly. The ProjectBuilder invokes the linker automatically.

9.1. Calling the linker from the command line

ALINK can be started from the command line and all files to be linked must be specified on the command line. Alternatively a link script which controls the link process can be used. The general syntax for the call is:

```
ALINK [<options>] <OBJ> [<LIB>] [<RES>] [/OUT:<EXE>]
```

or

```
ALINK @<LinkScript1> [@<LinkScriptN>]
```

When ALINK is called, one or more OBJ files must be specified as input files. The linker creates an EXE file as an executable program from the OBJ files. By default, the name of the first OBJ file is used as the name for the resulting EXE file. Input file names are separated by blank spaces and may appear in any order on the command line. ALINK uses .OBJ as default file extension. This means that OBJ files can be specified without extension to the linker. ALINK uses extensions only for searching files. It does not make any assumptions about file contents from file extensions.

In addition to OBJ files, ALINK accepts LIB files and one RES file as input files. LIB files are used for creating DLL files (see Creating DLL files) while a RES file contains binary resources, such as bitmaps or icons, which must be available at runtime (see The Alaska Resource Compiler - ARC.EXE).

Input files for the linker can also be listed in an ASCII file *<LinkScript>* which must be specified with a preceeding @ sign on the command line. The linker reads this file and links all files listed therein to an executable output file. The linker rounds the value <n> to a multiple of 64kB.

9.2. Linker options

The linker options pass information to ALINK specifying how an excutable file is created. It uses keywords which are always preceded by a slash. Linker options can be specified at any place on the command line, and some of the key words can be abbreviated.

/BASE:<n>

With the option /BASE:<n>, the linker sets the specified base address for loading the executable file. Default values are 0x400000 for EXE files and 0x10000000 for DLL files. If the operating system is not able to load a file at the specified location, it calculates a new base address and relocates the program. Information about base addresses can be obtained by using the /MAP option. Base addresses are then listed in a MAP file.

/DE[BUG]

With option /DEBUG, the linker imports debug information, such as symbol names and line numbers, into the EXE file. The EXE file can still be executed outside of the debugger. Within the debugger, this information is available only when the application has been linked with the /DEBUG option.

Important: To receive debug information from the compiler, PRG files must be compiled with the /b switch. A debug-enabled EXE file can be created only by using the compiler option /b along with the linker option /DEBUG.

/DEFAULTLIB:<file>

This option specifies an additional LIB file in which the linker is to search for resolving external references. The linker first searches the libraries specified on the command line, then the /DEFAULTLIB libraries and finally those named in OBJ files.

/DLL

With this option, the linker creates a DLL file instead of an EXE file. All OBJ files to be linked to a DLL must be compiled using the /dll switch of the compiler.

Note: When using the option /DLL, not only OBJ files but also the export definition file (EXP file) must be specified to the linker (see Creating DLL files).

/MAP[:<file>]

The /MAP option causes the linker to create a MAP file. This file contains information about the executable file. Optionally, the MAP file name can be specified with *<file>*. If it is omitted, the output file name is used for the MAP file.

/NOD[EFAULTLIB]

This option prevents ALINK from searching default import libraries which resolve external references, such as the file XPPRT1.LIB, for example. When this option is used, the linker only uses the LIB files specified on the command line.

/NOL[OGO]

This option suppresses the display of the program logo and version number.

/OUT:<file>

The name of the executable file can be defined with this option. By default, the file name is created from the first OBJ file specified to the linker.

/PM[TYPE]:VIO|PM [<major>[.<minor>]]

/SUBSYSTEM:CONSOLE|WINDOWS [<major>[.<minor>]]

The /SUBSYSTEM and /PMTYPE options have the same meaning for the linker. They define the application type. The default is VIO or CONSOLE, respectively, which indicates a text mode application. If PM or WINDOWS is specified, the linker creates a GUI application.

Optionally, the minimum version number of the operating system required for an Xbase^{++} application to be executed can be specified. If *<major>* is set to 4 and *<minor>* to .1, for example, an Xbase^{++} program could not be started on an operating system version 3.x but it would run on a 5.x version.

/ST[ACK]:<max>[,<min>]

This option sets the stack size of an Xbase^{++} application. *<max>* is a numeric value indicating the maximum stack size in bytes (default is 1 MB), while *<min>* optionally defines the stack size at program start. This means that the minimum stack size is *<min>* bytes and it can grow dynamically at runtime to a maximum size of *<max>* bytes.

/VERBOSE

The option /VERBOSE causes the linker to display additional information during the link process.

9.3. Environment variables for the linker

ALINK.EXE looks for the environment variable "ALINK" and uses its contents as default options. An example of a definition for the environment variable is:

```
SET ALINK=/DEBUG /PMTYPE:PM
```

This creates a debug-enabled GUI application by default. All options specified on the command line are processed after the SET ALINK= options. Therefore, command line options override those set with the environment variable.

In addition, the linker uses SET LIB= as a search path for OBJ and LIB files. ALINK searches for OBJ and LIB files in the following order:

1. The directory specified as part of a file name.
2. The current directory.
3. All directories defined in the environment variable SET LIB=.

If an OBJ or LIB file is not found, ALINK issues an error message and terminates the link process.

9.4. Creating DLL files

Dynamic linked libraries (DLL files) form the basis of the operating system and should generally be considered as part of an application during program development under Xbase^{++}. This chapter discusses how to build DLL files and uses as an example the following three procedures, each of which is assumed to be programmed in a separate PRG file:

```
** File MAIN.PRG **                 // This file is used to
   PROCEDURE Main                   // create an EXE file.
      SayHello()                    // Procedures are contained
      SayHi()                       // in a DLL file.
   RETURN

** File SAYHELLO.PRG **             // These two files are used
   PROCEDURE SayHello               // to create a DLL file.
      ? "Hello world"
   RETURN
```

```
** File SAYHI.PRG **
   PROCEDURE SayHi
      ? "Hi folks"
   RETURN
```

The Xbase^{++} ProjectBuilder provides the easiest way for creation of a DLL file since it can perform all necessary steps automatically. It is only required to create a project file which contains separate sections for EXE and DLL files. An appropriate project template for the ProjectBuilder can look like this:

```
// File: PROJECT.XPJ
[PROJECT]
    ROOT

[ROOT]
    MAIN.EXE
    MYFUNCS.DLL

[MAIN.EXE]
    MAIN.PRG
    MYFUNCS.LIB

[MYFUNCS.DLL]
    SAYHELLO.PRG
    SAYHI.PRG
```

The entire project consists of one EXE and one DLL file. The template lists the coresponding PRG source files in two separate sections. The section for the EXE file includes the import library MYFUNCS.LIB which is necessary for using DLL functions. This template must be expanded by calling PBUILD with the /g option and a second call - without the /g option - finally creates both EXE and DLL with its import library file.

If a DLL file is to be created without the ProjectBuilder, using a Make utility, for example, a total of five different steps must be performed:

1. Compile all PRG files required for the DLL file with the /dll compiler switch.

2. Create a file with the definitions for the modules in the DLL file (DEF file). This file contains a list of all functions or procedures which are exported from the DLL file and can be imported to an EXE file. This task is accomplished by the utility progam XPPFILT.EXE.

3. Create an import library (LIB) and an export file (EXP) from the DEF file. This is done by the utility program AIMPLIB.EXE.

4. Link OBJ files with the EXP file to a DLL.

5. To be able to use the new DLL file with a newly created EXE file, the import library (LIB) must be linked to the EXE file.

The following example describes these five steps. It uses the three PRG files MAIN.PRG, SAYHELLO.PRG and SAYHI.PRG:

Step 1: Compiling

Files for an EXE file or a DLL file must be compiled differently:

```
XPP main     /q /b

XPP sayhello /q /b /dll
XPP sayhi    /q /b /dll
```

Three OBJ files are created. The MAIN.OBJ file can only be linked to an EXE file and the files SAYHELLO.OBJ and SAYHI.OBJ can only be linked to a DLL file.

Step 2: Create the DEF file

To use a DLL file, all functions or procedures which can be called from outside the DLL must be known (export definitions). Definitions for exported functions or procedures are listed in a DEF file which is created by the utility program XPPFILT.EXE. It generates a DEF file from a list of OBJ files which are to be linked to a DLL:

```
xppfilt sayhello.obj sayhi.obj /f:myfuncs.def
```

XPPFILT.EXE creates the file MYFUNCS.DEF which contains all information for creating the file MYFUNCS.DLL. The example DEF file looks as follows:

```
01:   LIBRARY myfuncs INITINSTANCE TERMINSTANCE
02:   DATA MULTIPLE NONSHARED READWRITE LOADONCALL
03:   CODE LOADONCALL
04:
05:   EXPORTS
06:
07:   ;From object file: sayhello.obj
08:    SAYHELLO
09:
10:   ;From object file: sayhi.obj
11:    SAYHI
```

In the DEF file, comment lines start with semicolons. All other lines contain statements for the linker. The statement LIBRARY declares the file name of the DLL file and indicates whether the initialization routines in the DLL file are executed only once during loading or each time a process requires the DLL file. All DLL files created with Xbase^{++} must execute their initialization routines for each process (each program) and INITINSTANCE (line 1) must always be specified.

When multiple Xbase^{++} programs access the same DLL file, they can only share the program code and not the variables declared in it. All data in a DLL file must be given the attributes MULTIPLE NONSHARED READWRITE. This is done using the statement DATA (line 2).

The option LOADONCALL specified in the statements DATA and CODE specify that the DLL file is loaded into memory only when a module contained in the DLL file is executed. The alternative is the option PRELOAD, which specifies that loading takes place at the start of the EXE file (lines 2 and 3).

Following the EXPORTS statement (line 5) all identifiers (names) for exported functions and procedures must be listed. Each identifier must appear on a line by itself. Only the functions or procedures specified following the EXPORTS statement can be called from an EXE file.

Note: If classes are declared in a DLL file, only the class names must be listed, not the method names declared for a class. The class name is also the name of the class function and this must be defined as exported.

Step 3: Create the import library

The DEF file is used by the utility program AIMPLIB.EXE to create an import library (LIB file) and an export file (EXP file):

```
aimplib myfuncs.def
```

As a result, the files MYFUNCS.LIB and MYFUNCS.EXP are created. The LIB file contains information about what can be imported by an EXE file and the EXP file defines what is exported from a DLL file.

Step 4: Create the DLL file

When the EXP file exists, the DLL file can be created by the linker. All OBJ files plus the EXP file must be specified:

```
ALINK /DLL sayhello.obj sayhi.obj myfuncs.exp /OUT:myfuncs.dll
```

OBJ files must be linked using the /DLL option. The name of the DLL file is defined using the /OUT option.

Step 5: Create the EXE file

The last step creates the executable EXE file. All import libraries containing references to additional DLL files must be specified to the linker:

```
ALINK main.obj myfuncs.lib
```

This call to ALINK creates the executable file MAIN.EXE as text mode application. It contains no code from the DLL file, but references the dynamic library MYFUNCS.DLL. The code from this file is loaded when a function contained in the DLL is called from MAIN.EXE.

9.5. The utility program AIMPLIB.EXE

The program AIMPLIB.EXE creates import libraries and export files from export definition files (DEF files). These files are required for the creation of DLL files to be linked statically to an EXE (see the previous section). AIMPLIB accepts the following command line options:

/? | /h	Displays information about command line options.
/coff	Creates an import library in Common Object File Format (COFF). This is the default option for Windows platforms.
/omf	Creates an import library in Object Module Format (OMF). This is the default option for the OS/2 platform.
/o:<file>	Specifies the file name *<file>* for the import library.
/q	Suppresses screen output while the program is running (quiet mode).

10. The Alaska Resource Compiler - ARC.EXE

In GUI applications, it is a common practice to use external resources for displaying graphic information which cannot be included in PRG source files, such as bitmap images, for example. One way of supplying external resources to an application is by linking the resources to the executable file. In this way, it is guaranteed that resources are available for the program at runtime.

External resources must exist in a binary format for the linker when it binds them to an executable file. The conversion to binary format is the task of the resource compiler ARC.EXE which uses a description file for external resources (ARC file) to create a binary resource file (RES file).

10.1. Declaring external resources - The ARC file

When an Xbase^{++} application requires external resources, they are declared in an ARC file. This is a file in ASCII format which is compiled to a binary resource file (RES file) by the resource compiler ARC.EXE. External resources are bitmaps, icons and pointers. In an application program, they are identified by a numeric ID. Therefore, the ARC file must provide data about the resource type, a numeric ID for each resource and the name of the file that contains the resource:

```
*
* TEST.ARC
*

/* Declaration of different
   resource types
*/

BITMAP      110 = "Logo.bmp"         // Bitmap
ICON        120 = "Folder.ico"       // Icon
POINTER     130 = "Arrow.ptr"        // Pointer

* EOF
```

These lines show the important syntactical elements for declaring resources in an ARC file. Declaration begins with a keyword indicating the resource type. It is followed by a numeric ID for identifying the resource within an application. After the ID, an equal sign must appear and the name of a file enclosed in double quotes completes the declaration. Comments can be included in the ARC file (as in a PRG file) by using the characters /* and */ or a double slash for inline comments. In addition, the asterisk indicates a comment line when it is the first character in a line.

10.2. Directives for ARC.EXE

Instead of numeric IDs, #define constants can be used in an ARC file just as in a PRG file. This has the ovious advantage that resources can be identified in both ARC and PRG files using the same constants. To accomplish this, the constants are defined in a CH file. The resource compiler understands the directives ***#inlude***, ***#define***, ***#ifdef***, ***#ifndef***, ***#else*** and ***#endif***, and treats them in the same way as the Xbase++ compiler. Therefore, it is possible to declare resources in the following way:

```
** CH file: BITMAPID.CH                  // Include file defines
                                         // constants for
#define ID_BMP_LOGO          10          // XPP.EXE and ARC.EXE
#define ID_BMP_BACKGROUND    11
#define ID_BMP_NEXT          12
#define ID_BMP_PREVIOUS      13

** EOF

-----------------------------------------------------------------

** ARC file: TEST.ARC                    // Declaration of resources

#include "BitmapID.ch"                   // Include CH file

BITMAP                                   // Keyword introduces
   ID_BMP_LOGO       = "Logo.bmp"        // a block of resources
   ID_BMP_BACKGROUND = "Backgrd.bmp"     // of the same type
   ID_BMP_NEXT       = "Next.bmp"
   ID_BMP_PREVIOUS   = "Prev.bmp"

   #ifdef __OS2__                        // Conditional
      100 = "\bitmaps\os2\test.bmp"      // compiling due to
   #else                                 // implicit constant
      100 = "\bitmaps\w32\test.bmp"
   #endif

** EOF
```

This example shows the code for a CH file which defines constants for the numeric resource IDs. The constants are used in the ARC file since ARC.EXE can resolve the #include directive.

Only bitmap resources are declared in the example. The keyword BITMAP is written in a separate line and is followed by a block of resource declarations of the same type. ARC supports this syntactical form for declaring resources which results in better readability of the ARC file. If a keyword is written on a line by itself, all following lines declare the same resource type without repeating the keyword.

The last lines in the example demonstrate how operating system-specific resources can be declared in one and the same ARC file. The directives *#ifdef*, *#else* and *#endif* are used, thus allowing for a conditional compilation. ARC.EXE uses the same implicit #define constants as the Xbase^{++} compiler: __OS2__ and __WIN32__.

10.3. Options for ARC.EXE

The resource compiler is started on the command line using the following syntax:

```
ARC [<options>] <file1> [<fileN>]
```

ARC.EXE creates one RES file from all ARC files specified on the command line. If the #include directive appears in an ARC file, the resource compiler searches for the include files in the current directory and in the directories listed in the INCLUDE environment variable. The files containing the external resources - like bitmaps or icons - are searched in the current directory and in the directories listed in the XPPRESOURCE environment variable.

The resource compiler accepts the following command line options:

/? | /h	Displays information about ARC options.
/d<id>[=<val>]	The /d option specifies the #define constant <*id*> to the resource compiler on the command line. The #define constant is valid within the ARC file. Optionally, the constant can be assigned the value <*val*>.
/ga	When the /ga option is used, all literal character strings in the ARC file are converted to ANSI before the resource compiler creates the RES file.
/go	When the /go option is used, all literal character strings in the ARC file are converted to OEM before the resource compiler creates the RES file.
/i:<path>	The /i option specifies an additional search directory <*path*> for the resource compiler to use when locating #include files. Normally, ARC.EXE only searches for these files in the directories specified by the INCLUDE environment variable.
/o:<name>	Normally, the resource compiler creates a RES file which has the same file name as the ARC file it is created from. The /o switch is used to rename the resulting RES file to <*name*>.
/q	Suppresses screen output while compiling (quiet mode).
/v	The switch /v activates the verbose mode of the resource compiler.

11. Basics of Database Programming

This chapter describes the basics of database programming. It explains fundamental aspects as well as major terms used in the context of databases and database access.

11.1. What is a database

Theoretically, the word "database" refers to the entire set of data belonging to one problem domain. This data can be divided into many files. In the context of xBase dialects "database" is used when discussing individual files that exist in the DBF file format. For this historical reason, the word "database" is used in this documentation as a synonym for a single DBF file.

The DBF file format is the basis for all xBase dialects and allows management of extensive data sets in a straightforward manner. The data within each DBF file is organized in table form. Each column corresponds to a database field (field, for short) and each row corresponds to a "data record" (record, for short). The title (or top row) contains the field names identifying the fields. The following table illustrates how data is organized within a DBF file:

CUSTNO	LASTNAME	FIRSTNAME	PHONE
40001	King	Michael	(609)423-4567
40002	Fisher	Fred	(614)713-4578
50013	Anderson	Robert	(819)567-9832
50021	Long	Barbara	(402)715-4321
50043	Baker	Christine	(517)454-3356
50057	Kemper	Joseph	(414)234-5678
50100	Smith	Richard	(303)614-4321

The DBF file (table) in this example contains seven records (rows). A single record includes four fields (columns). Each field contains specific information about a customer.

In a DBF file, the structure of the table is defined along with the data. The table definition includes the field names (column headings), field lengths (column widths), data types and number of decimal places for numeric fields. A file contains strongly typed data, meaning the data in a column always has the same data type. The DBF file format allows a total of five different data types to be stored: character, date, logical, memo and numeric.

The initial definition of the structure of a DBF table can also be stored in a DBF table. To be used in this way, the table must contain the field name, data type of the field, field length and the number of decimal places (a total of four columns).

The definition of the table structure for the above example would be the following:

FIELD_NAME	FIELD_TYPE	FIELD_LEN	FIELD_DEC
CUSTNO	C	8	0
LASTNAME	C	20	0
FIRSTNAME	C	20	0
PHONE	C	15	0

This table contains all the information necessary to describe the structure of the DBF file. Each record contains the definition for one field. The column headings in this table are the field names of what is called a "structure extended" DBF file. A structure extended DBF file contains the field definitions for another DBF file. The field definitions comprise the definition of the structure of a DBF file, and are the starting point for database usage.

11.2. Creating a database

A DBF file can be created only from a structure definition. In Xbase^{++} the functions and commands listed in the next table can be used to create this definition:

Functions and commands for the creation of databases

Function	Command
DbCreateExtStruct()	CREATE
DbCreateFrom()	CREATE FROM
DbCopyStruct()	COPY STRUCTURE
DbCopyExtStruct()	COPY STRUCTURE EXTENDED
DbCreate()	

The first four functions and commands in this table exist only for compatibility reasons and should no longer be used. Creating a DBF file in Xbase^{++} should be done using the function DbCreate(). DBCreate() takes as a parameter the structure definition for the DBF file in the form of a two dimensional array. This avoids the round-about mechanism of using a structure extended file. The following code demonstrates the optimal use of the function DbCreate():

```
cFileName   := "CUST.DBF"
aStructure := { { "CUSTNO"     , "C",  8, 0 }, ;
                { "LASTNAME"   , "C", 20, 0 }, ;
                { "FIRSTNAME"  , "C", 20, 0 }, ;
                { "PHONE"      , "C", 15, 0 }  }

DbCreate( cFileName, aStructure )
```

The file name as well as the structure for the DBF file are stored in variables passed to the function DbCreate(). The file structure is defined in a two dimensional array. All fields in this example have the "character" data type, identified using the letter "C" (character). The length of the fields are 8, 20, 20 and 15 characters, respectively, and all fields have 0 decimal places.

A field name may contain a maximum of 10 characters. The same rules apply to field names that apply to variable names. The first letter must be an alphabetical character or an underscore and all other characters must be alphanumeric or an underscore.

The data type of a field is defined by a single letter. The letters "C" (character), "D" (date), "L"(logical), "M" (memo) and "N" (numeric) are recognized.

The maximum length of a field depends on the data type:

Character ("C")	Maximum 64 kBytes
Date ("D")	Always 8 Bytes
Logical ("L")	Always 1 Byte
Memo ("M")	Always 10 Bytes
Numeric ("N")	Maximum 19 Bytes

A memo field always occupies 10 bytes, but can store an unlimited number of characters. This is because the contents of memo fields are stored in a separate file called a DBT file.

Numeric fields can store numbers with a maximum length of 19 places. The decimal and the prefix each take up one byte. The largest number which can be stored is 2^32-1 and the maximum number of decimal places is 15.

11.3. Saving data

After a DBF file is created using DbCreate(), it must be opened before data can be stored in it or read from it. The following table gives an overview of the functions and commands that can be used for the simple database operations "open", "close", "create record" and "change field contents".

Functions and commands for simple database operations

Function	Command	Description
DbUseArea()	USE	Opens database
DbAppend()	APPEND BLANK	Appends new record
FieldPut()	REPLACE	Changes contents of fields
DbCloseArea()	CLOSE	Closes database

Whether database operations are programmed using functions or commands is simply a matter of the personal preference of the programmer. At runtime of a program there is no difference between a database operation that is programmed as a command and one programmed as a function. In many cases the command syntax is easier to program with. The following program code shows operations using command syntax which can be contrasted to the next example which uses function syntax.

```
USE Customer                                 // open database
APPEND BLANK                                 // append record
REPLACE CustNo WITH "    50112"      , ;     // enter data in
     LastName  WITH "Miller"         , ;     // fields of the new
    FirstName  WITH "Karl"           , ;     // record
        Phone  WITH "(713)517-6554"
CLOSE Customer                               // close database
```

In the example, a new record is added to a database containing customer information. The same operations appear below using the function syntax:

```
DbUseArea(, , "Customer")                    // open database
DbAppend()                                   // append record
FieldPut( 1 , "    50112")                   // enter data in
FieldPut( 2 , "Miller" )                     // fields of the new
FieldPut( 3 , "Karl" )                       // record
FieldPut( 4 , "(713)517-6554" )
DbCloseArea()                                // close database
```

In this example, the customer data is written into the database using the function FieldPut() which identifies the field by its ordinal position. In doing this, the program code loses clarity. A better solution, in this case, is to address the fields by the alias name FIELD and perform direct assignments:

```
DbUseArea(, , "Customer")               // open database
DbAppend()                              // append record
FIELD->CustNo := "   50112"             // enter data in
FIELD->LastName := "Miller"             // fields of the new
FIELD->FirstName  := "Karl"             // record
FIELD->Phone  := "(713)517-6554"
DbCloseArea()                           // close database
```

Along with these elementary database operations, Xbase++ provides many functions that can be used to obtain information about the database or individual fields. A database must be open in order for the functions in the next table to be used:

Functions that return information about a database

Function	Description
Header()	Returns length of the file header in bytes
RecSize()	Returns length of a record in bytes
RecCount()	Returns number of records in the file
LastRec()	Returns number of records in the file
LUpdate()	Returns date of the last write access

These functions can be used to calculate the file size of a database (the functions RecCount() and LastRec() both provide the same value). The formula for calculating the size of a database is:

```
(RecSize() * LastRec()) + Header() + 1
```

This formula is frequently used, along with the function DiskSpace(), during backup routines to determine whether a DBF file fits on the target drive where it is to be copied.

Information about fields are very important in database programming. Not only are the contents of a field used when processing data, but the data type, ordinal position within the DBF file and the name of a field can also be used. For example, this information becomes useful when writing generic, or data-driven read/write routines.

Functions returning information about fields in a database

Function	Description
FCount()	Returns number of fields in the file
FieldName()	Returns field name based on ordinal position
FieldPos()	Returns ordinal position of field based on field name
FieldGet()	Returns contents of field based on ordinal position
Type()	Returns data type of field based on field name
DbStruct()	Returns file structure as a two dimensional array

The following code presents an example using some of the functions from this table. This code includes two user-defined functions. One of these reads all fields of a record into an array and the other writes the array back into the file. This a common programming technique for buffered editing of records and is frequently used in programs designed for multi-user operation in a network environment:

```
PROCEDURE Main
   LOCAL aRecord := ReadRecord()     // read record

   // <edit data>

   IF RLock()                        // lock record
      WriteRecord( aRecord )         // write record
      DbUnlock()                     // release record
   ENDIF
RETURN

FUNCTION ReadRecord()                // read record
RETURN AEval( Array( FCount() ), {|x,i| x:= FieldGet(i) },,, .T. )

FUNCTION WriteRecord( aRecord )      // write record
RETURN AEval( aRecord, {|x,i| FieldPut(i, x) } )
```

11.4. Work area and Alias

In database programming it, is the rule, rather than the exception, that an application program uses multiple databases (DBF files) at the same time. Access to different databases is accomplished by means of work areas at runtime of a program. A work area is identified by a numeric index and has two different states, *free* and *used*. When a database is opened in a work area with the USE command, the work area is used. Only one database can be open in a work area and access to a database can occur only in the selected -or current- work area. In database programming, work areas play a central role since they encapsulate access to databases.

When a database is opened, a work area gets a symbolic identifier: the alias name. If not specified explicitly, the alias name is implicitly created from a database's file name. Thus, a work area has three important attributes and there are three functions to return this information.

Functions for work areas

Function	Description
Select()	Returns the number of a work area
Used()	Determines whether a work area is used
Alias()	Returns the alias name of a work area

Information from different work areas can be retrieved using the alias operator (->) and the number of a work area, or its alias name. Also, fields of multiple databases are accessed via alias reference. Referencing work areas with aliases is a fundamental operation and frequently used in programming with multiple databases, or work areas, respectively. The following lines of code show some basic operations:

```
PROCEDURE Main
   ? Select()                          // result: 1
   ? Used()                            // result: .F.
   ? Alias()                           // result: "" (null string)

   USE Customer                        // Open database without alias
   USE Invoice  ALIAS Inv  NEW         // Open databases in new work
   USE Orders   ALIAS Ord  NEW         // areas with alias names

   ? Select()                          // result: 3
   ? Used()                            // result: .T.
   ? Alias()                           // result: Ord

   ? (1)->(Alias())                    // result: Customer
   ? (2)->(Alias())                    // result: Inv

   ? Customer->Lastname                // result: Miller
   ? (2)->InvDate                      // result: 06.12.1994
   ? Ord->(Select())                   // result: 3
RETURN
```

This program demonstrates the most important aspects for using multiple databases. At the beginning of the Main procedure, work area number 1 is selected; it is the current work area. Since no database is initially open, the function *Used()* returns the value .F. (false) and the alias name of the first work area is a null string (""). Only after a database is opened with the USE command does the work area receive an alias name. The return value .T. (true) of the *Used()* function then indicates the current work area to be used. When the USE command is programmed with the NEW option, it first selects a new work area and then opens a database. Therefore, the return value of the *Select()* function is now 3 and no longer 1 as in the first call. Three databases are open and the third work area is current. Its alias name is "Ord".

When a function is called without alias reference, it is executed in the current work area. An alias reference causes a function to be executed in the corresponding work area (more precisely, the corresponding work area temporarily becomes the current one, then the function is executed and the previous work area is selected again). The function call *(1)->(Alias())* returns the alias name of the first work area. It is the string "Customer" which is implicitly created from the file name.

The example also shows syntax differences which are required for alias references to database fields (or symbols) and function calls (or expressions). If an expression appears on the right side of the alias operator, it must be written in parentheses. A symbol is programmed without parentheses. On the left side of the alias operator, parentheses must be used when a number or a variable is used instead of the literal alias names. Therefore, alias references can be coded in different ways:

```
cAlias := "Customer"
nArea  := 1

xValue := Customer->Lastname      // Alias reference to field variable
xValue := (1)->Lastname
xValue := (cAlias)->Lastname
xValue := (nArea)->Lastname

xValue := Customer->(Select())    // Execute an expression in
xValue := (1)->(Select())         // a work area
xValue := (cAlias)->(Select())
xValue := (nArea)->(Select())
```

Alias names of work areas are guaranteed to be unique, just as work area numbers are. If the same database file is opened multiple times in different work areas, each work area will receive a different alias name but reference the same physical database.

```
USE Customer NEW
USE Customer NEW
USE Customer ALIAS Cust NEW
USE Customer ALIAS Cust NEW

? (1)->(Alias())                      // result: Customer
? (2)->(Alias())                      // result: Customer_2
? (3)->(Alias())                      // result: Cust
? (4)->(Alias())                      // result: Cust_4
```

When an attempt is made to use the same alias name more than once, Xbase^{++} implicitly creates an unique identifier by adding an underscore and the work area number to the original alias name.

11.5. The work space of Xbase++

The concept of work areas exists in every xBase dialect. It is extended in Xbase++ by a work space which provides a higher level of abstraction for work areas. The concept of work spaces can be described as follows:

1. A work space contains work areas.

2. The number of work areas inside a work space is limited to 65.000.

3. A work space defines the current work area. All database operations programmed without alias reference are executed in this work area.

4. Selecting the current work area is a work space operation. Each work space always has one current work area.

5. Each thread has at least one work space.

A work space is a container for work areas and allows multiple databases at runtime to be opened. Access to database fields in a program follows a hierarchical pattern described below:

```
Work space              One work space in each thread
 |
 +- Work area           A maximum of 65.000 work areas per work space
     |
     +- Alias name      References an open database
         |
         +- Field name  References a database field (field variable)
```

A work space and contained work areas always exist at runtime of a program, regardless of whether databases are open or not. The fundamental operation in a work space is selection of the current work area. As long as a work area is not used, it can be selected by its number. A number between 1 and 65.000 is passed to the command *SELECT* or function *DbSelectArea()* in order to select a particular work area as the current one. As an alternative, the value 0 (zero) can be specified. This causes the next free work area with the smallest number to become current. For example:

```
PROCEDURE Main
   USE Customer
   ? Select()                        // result: 1

   SELECT 100
   ? Select()                        // result: 100
   SELECT 0
   ? Select()                        // result: 2
RETURN
```

At program start, work area 1 is selected and the database CUSTOMER.DBF is opened in the example with the USE command in the first work area. Then, the work area 100 is selected and becomes the current one. Finally, the SELECT 0 command selects the next free work area. It has the ordinal position 2.

When a database is opened in a work area. it can be selected not only by its number but by the alias name. The alias name is a symbolic name that makes programming with multiple databases much easier:

```
PROCEDURE Main
   USE Customer
   USE Invoice NEW
   USE Parts   NEW

   ? Select()                        // result: 3
   ? Alias()                         // result: Parts

   SELECT 2
   ? Alias()                         // result: Invoice

   SELECT Customer
   ? Select()                        // result: 1

   DbCloseAll()                      // Close all databases
RETURN
```

Three databases are opened in this example in three different work areas. It demonstrates how to select a database using a number or an alias name. At the end, all databases are closed.

Operations in one work space

The function call *DbCloseAll()* is an example for functions that perform operations in a work space. It affects all used work areas, not the free ones. There is a set of functions in Xbase^{++} that operates on a single work space and performs operations with a single work area or all used work areas.

The following table lists these functions:

Functions and commands for one work space

Function	Command	Description
Affects one work area		
DbSelectArea()	SELECT	Selects the current work area
Affects all used work areas		
DbCloseAll()	CLOSE ALL	Closes database files
DbCommitAll()	COMMIT	Writes buffers permanently to disk
DbUnlockAll()	UNLOCK	Releases record and file locks
WorkspaceEval()		Evaluates a code block in all used work areas
WorkspaceList()		Returns alias names of all used work areas in an array

The *WorkspaceEval()* function listed in the table is the most powerful one, since it evaluates a code block in all used work areas. All *Db...All()* functions could be emulated with this function and new functions for all work areas can be programmed.

Operations between two work spaces

In Xbase^{++}, a work space is bound to a thread. As a consequence, work areas are also thread local resources, because they are contained in a work space. This implies in turn that database access is limited to the thread where a database is opened. However, a multi-threaded program must be able to access databases from different threads. This can be achieved either by opening the same database in different threads or by exchanging the reference to an open database between threads or work spaces, respectively. The functions listed in the next table serve the latter purpose:

Functions for multiple work spaces

Function	Description
DbRelease()	Transfers the alias of a work area into the Zero space
DbRequest()	Transfers an alias from the Zero space into a work area

In Xbase^{++}, an alias name is the reference to an open database. The alias name of a work area can be exchanged between two work spaces. Transferring alias names involves a virtual work space, which is called the Zero space.

The two functions *DbRelease()* and *DbRequest()* are used for exchange:

```
USE Customer
? Used()                        // result: .T.

DbRelease()
? Used()                        // result: .F.

DbRequest()
? Used()                        // result: .T.

CLOSE Customer
```

After calling *DbRelease()*, the current work area is free. However, the database file is still open. It is the reference to the database (the alias name) that no longer exists in the current work space. It is transferred to the Zero space, and can be transferred back to a work area of a work space by calling the *DbRequest()* function. When *DbRequest()* is finished, a work area has received an alias name and is used again. It is important to understand that the database file remains open during these operations. Only the reference to the open file changes its place from the current work space to the Zero space back to the same or to another work space.

11.6. Record pointer and database fields

When a work area is used, all data that is stored in a database file can be accessed. After a database is opened, the record pointer initially points to the first record. At runtime, the field variables of a program then contain the values that are stored in the first record. This means that a program can only access fields of the current record. In order to access data stored in different records, the record pointer must be moved. Database navigation must occur in a work area:

```
USE Customer
? Recno()                        // result: 1
? Customer->Lastname             // result: Miller

GOTO 10                          // select record #10

? Recno()                        // result: 10
? Customer->Lastname             // result: Brown
```

This example demonstrates that the contents of field variables change when the record pointer is moved. Records are selected by navigating the record pointer and this is the only way to access all data stored in a database file. The next table lists important functions used in the context of database navigation:

Functions and commands for database navigation

Function	Command	Description
Bof()		Beginning of file is reached?
Eof()		End of file is reached?
DbSkip()	SKIP	Moves the record pointer
DbGoTo()	GOTO	Moves the record pointer to a particular database record
DbGoTop()	GO TOP	Moves the record pointer to the first database record
DbGoBottom()	GO BOTTOM	Moves the record pointer to the last database record
DbGoPosition()		Moves the record pointer on a percent basis
DbPosition()		Returns record pointer position as a percent value
RecNo()		Returns the record pointer

The functions *Recno()* and *DbSkip()* are by far the most important functions for database navigation. The former returns the record pointer and the latter changes it. Also, the two functions *Bof()* and *Eof()* are used very often. They indicate whether the record pointer has reached the boundaries (top or bottom) of a database file. The following code example shows a typical construction used to get information from all records of a database file:

```
USE Customer                                 // Open database

DO WHILE .NOT. Eof()                         // Exit loop at end of file
                                             // Display single fields
   ? Recno(), FIELD->Firstname, FIELD->Lastname
   DbSkip(1)                                 // Next record
ENDDO

CLOSE Customer                               // Close database
```

12. The Xbase++ DatabaseEngine

This chapter describes the concepts of the Xbase++ DatabaseEngine and many aspects of its use. Contrary to other Xbase development environments, there is not a single monolithic database driver in Xbase++ (even one that is replaceable such as a Replaceable Database Driver, or "RDD"). Instead, Xbase++ uses the concept of a "database engine" ("DBE"), which consists of individual components. These components are objects which make services (methods) available for data and file management. The individual components can be assembled into a compound DBE. With this modular concept the programmer has the flexibility to access the various database models available today, as well as any model in the future.

12.1. Basics of DatabaseEngines

The architecture of the Xbase++ DatabaseEngines (DBE) is designed to be applicable to any database model. This includes both record oriented databases (for example, xBase databases) as well as set oriented and relational databases (for example, SQL databases). A database can be described abstractly as having four components:

DATA component	This component describes the database model or how the data itself is stored. "Database" here can be considered analogous to a "table" in the relational model or a "structure" in the hierarichal database model.
ORDER component	This component makes a mechanism available to allow the definition of logical sort orders based on expressions. This allows leaving the pre-existing physical order of the tables intact.
RELATION component	This component defines the relationships between data. It links tables either physically or logically based on expressions. The task of this component can be considered analogous to the "JOIN operator" in the relational model or the "Set-pointer" in the file-based network database model.
DICTIONARY component	This component describes the database structure and/or integrity rules.

A DatabaseEngine represents one or more of these four components and defines how they interact with the underlying database. The DBE itself is contained in a DLL file and is loaded at runtime of a program. For example, the file DBFDBE.DLL contains the DBE which provides the component for managing DBF files (DATA component). The file NTXDBE.DLL contains a DBE which represents a component for orders (ORDER component) that is valid for record oriented databases (for example, DBF files). This DBE manages index files in the NTX format. The DBFDBE and the NTXDBE are DatabaseEngines which each contain only one component for data and file management. They are each designated as a "component DBE".

Two component DBEs which provide different components can be assembled into one new DatabaseEngine which then controls both components and combines all the characteristics of the component DBEs within the new DBE. Such an assembled DatabaseEngine is called a "compound DBE". A compound DBE exists only in memory and is not saved in a file on the hard disk. Component DBEs, on the other hand, are stored in files.

Compound DBEs are generally needed when programming with databases and must be created at the start of the program. The appropriate component DBEs are loaded into memory using the function DbeLoad() and the compound DBE is created using the function DbeBuild(). The following program example shows the basic approach:

```
IF .NOT. DbeLoad( "DBFDBE", .T. )          // load component DBE
   Alert( "DatabaseEngine DBFDBE not loaded", {"OK"} )
ENDIF

IF .NOT. DbeLoad( "NTXDBE", .T. )          // load component DBE
   Alert( "DatabaseEngine NTXDBE not loaded", {"OK"} )
ENDIF
                                           // create compound DBE
IF .NOT. DbeBuild( "DBFNTX", "DBFDBE", "NTXDBE" )
   Alert( "DBFNTX DatabaseEngine not created", {"OK"} )
ENDIF
```

In the example, two component DBEs stored in the files DBFDBE.DLL and NTXDBE.DLL are loaded into memory. From these two component DBEs a compound DBE is created with the name "DBFNTX". This compound DBE can manage DBF files (DATA component) as well as NTX files (ORDER component).

The example uses only two of the four basic functions available for programming using DatabaseEngines.

The next table lists all four functions:

Functions for the preparation of DatabaseEngines

Function	Description
DbeLoad()	Loads component DBE into memory
DbeBuild()	Creates compound DBE and defines it as current DBE
DbeSetDefault()	Defines default DBE for file operations
DbeUnLoad()	Removes DBE from memory

The function DbeLoad() loads a component DBE into main memory. Such a DBE can be loaded in two ways: hidden and visible. A component DBE which is loaded as "hidden" can only be used for creating compound DBEs and not directly used for data and file management.

```
DbeLoad( "DBFDBE", .F. )         // load component DBE as visible
USE Customer                     // open DBF file
Browse()                         // perform file operations
```

In this example, the component DBE is loaded as visible (.F. means not "hidden" here). Therefore the DBE can be used to open a DBF file, and execute file operations. The following example shows the opposite:

```
DbeLoad( "DBFDBE", .T. )         // load component DBE as hidden
USE Customer                     // runtime error
```

Here the component DBE is loaded as "hidden" and any attempt to use it for file management leads to a runtime error. The DBE can only be used to create a compound DBE using the function DbeBuild().

Note: The compound DBE "DBFNTX" is created by default each time an Xbase^{++} program is started. This means that all file and index commands and functions work with this DatabaseEngine by default. The DBFNTX DBE is created in the file DBESYS.PRG. The function DbeSys() contained in this file is called each time an Xbase^{++} program starts.

The DBFNTX DatabaseEngine consists of only two components and is able to mimic the database model of Clipper in how it handles DBF files and NTX files. The RELATION component is unique in this database model in that no DBE is required for the RELATION component. Also there is no DICTIONARY component, because this is not supported by the database model.

12.2. DatabaseEngines and programming language

For a complete understanding of DatabaseEngines, an explanation of the internal mechanisms which create the relationship between the programming language and the database model is required. As an example, the command USE and the function DbUseArea() open one or more files in a work area. The actual file opening is performed by a DatabaseEngine, in this case by the DATA component of the DBE. The command INDEX ON creates an index for a file open in the work area. This operation is also performed by a DBE, but the DBE uses the ORDER component.

Certain commands and functions exist within the confines of the programming language and can only be executed by a specific component of a DatabaseEngine. With these commands and functions, the request must be routed to the proper DBE capable of executing the command or function. This task is performed by an internal mechanism called the "Database Management Language Broker", abbreviated as DMLB. The DMLB routes requests from the program to a specific component of a DBE.

The DatabaseEngine DBFDBE, for example, is not capable of creating an index. To accomplish this, the DatabaseEngine NTXDBE must be loaded. Likewise, this DBE can not open a DBF file. In order for the command USE and the command INDEX ON to be executed, a compound DBE must be available that contains all components required by the program. The task of the DMLB is to direct requests from the program to the components of a compound DBE that are able to perform the required task. If no corresponding component is available, the DMLB attempts to perform the task itself based on its own abilities. If this is not successful, a runtime error occurs. This model provides the ability to exclude components that are not needed. For example, it is possible to write a program that does not use an index. In this case, the DatabaseEngine NTXDBE (more precisely, the ORDER component) does not need to be loaded.

The distribution of specific functionalities to component DBEs and the ability to make all functionalities dynamically available using a compound DBE offers much flexibility in data and file management. For example, an index can be created for a file that exists in SDF format (System Data Format). The NTXDBE (ORDER component) merely needs to be combined with the SDF DatabaseEngine (DATA component). The result is an SDFNTX DatabaseEngine that allows logical sorting of ASCII files that are in SDF format. The following program code illustrate this (Note: the example assumes that the compound DBE DBFNTX is already created).

```
USE Customer VIA DBFNTX              // open DBF file
COPY TO Address.txt SDF              // copy file into ASCII file
CLOSE Customer                       // (System Data Format)
```

```
DbeLoad( "SDFDBE", .T.)              // load SDF engine and
                                     // create compound DBE
DbeBuild( "SDFNTX", "SDFDBE", "NTXDBE" )

USE Address.txt VIA SDFNTX           // open ASCII file
INDEX ON Upper(Name) TO Temp         // create index
```

The fact that DATA components can be combined with ORDER components to create a compound DBE allows the functionality of various file operations to be available for different file formats. In the example a TXT file (ASCII file) is opened using the same command (USE) previously used to open a DBF file. Also database operations, like DbAppend() and DbSeek(), can now be performed on the TXT file. The addition of a new data record using DbAppend() is accomplished via the SDF component and the search for data in the TXT file using DbSeek() occurs via the NTX component.

There is, however, the limitation that it must be possible to perform the particular file operation with files of the file format. In the case of the SDF DatabaseEngine, deleting data records using the function DbDelete() is not possible. This function flags a record for deletion and there is no provision in the SDF file format for deletion flags. Consequently, calling the function DbDelete() for a file managed by the SDF DatabaseEngine would lead to a runtime error because this operation is not supported.

12.3. Determine information about DatabaseEngines

At runtime of a program it is sometimes necessary to determine information about loaded DBEs, work areas, or individual fields in work areas. Xbase^{++} provides the four functions listed in the following table for this:

Functions for information about DatabaseEngines

Function	Description
DbeList()	Determines which DBEs are loaded
DbeInfo()	Returns information about the current DBE
DbInfo()	Returns information about a work area
FieldInfo()	Determines information about a field based on ordinal position

For a deeper understanding of these functions, knowledge of the internal mechanisms used for opening files with USE or DbUseArea() is required. These mechanisms are not immediately recognizable at the language level. A file can only be opened when at least one DBE is loaded that provides the DATA component for the requested file format. Only after such a component is available can a file be opened in a work area, otherwise a runtime error occurs. Whether or not a DBE is currently available can be determined at runtime using the function DbeList(). DbeList() returns a two column array: the first column contains the names of DBEs and the second column contains logical values representing the "hidden-flag" (see previous section).

Opening a file using the current DBE creates an instance of the DBE which is called a "database object" (DBO). The DBO makes use of all the characteristics of the current DBE and it is actually the database object that opens and manages files. A DBO represents the work area where the files are opened.

It is helpful to distinguish between the DatabaseEngine, which provides the functionality that allows files to be opened in a work area and the database object which exists only while the file is open in the work area. The database object is created by the DatabaseEngine when the file is opened. It manages the file and is automatically discarded when the file is closed.

12.3.1. The function DbeInfo()

The function DbeInfo() provides information about the current DBE. Therefore, it can only be executed when a DBE is loaded. It is similar to the functions DbInfo() and FieldInfo() that can only be called when a file is open in the current work area. Otherwise a runtime error occurs.

As well as returning information, DbeInfo() also allows a DBE to be configured in specific ways. Several settings of a DBE are changeable and can be redefined using DbeInfo(). As an example, the default extension for files can be changed:

```
#include "Dmlb.ch"
#include "DbfDbe.ch"

DbeLoad( "DBFDBE", .T.)
DbeLoad( "NTXDBE", .T.)                // create the compound DBE
DbeBuild( "DBFNTX", "DBFDBE", "NTXDBE" ) // DBFNTX

                                     // default extension DBF->FBD
                                     // for database files
DbeInfo( COMPONENT_DATA , DBE_EXTENSION, "FBD" )
                                     // default extension NTX->XTN
                                     // for index files
DbeInfo( COMPONENT_ORDER, DBE_EXTENSION, "XTN" )

DbCreate( "Temp", { { "LName", "C", 20, 0 }, ;
                    { "FName", "C", 20, 0 }, } )

USE Temp                             // create database and
INDEX ON Field->LName TO TempA       // index files
INDEX ON Field->FName TO TempB
CLOSE Temp
                                     // display file names. Result:
```

```
AEval( Directory( "Temp*.*" ), ;        //   Temp.FBD
       {|a| QOut(a[1]) } )              //   TempA.XTN
                                        //   TempB.XTN
```

In this example, the default extensions for two different kinds of files are redefined using DbeInfo(). This affects the files managed by the DATA component of the DBFNTX DBE (the DBF file is created as a FBD file) and the files which are managed by the ORDER component (the NTX files are created as XTN files). All this happens in just two lines of the example:

```
DbeInfo( COMPONENT_DATA , DBE_EXTENSION, "FBD" )

DbeInfo( COMPONENT_ORDER, DBE_EXTENSION, "XTN" )
```

A special aspect of DbeInfo() is shown: if the function is called with parameters, the first parameter must always be a #define constant from the file DMLB.CH specifying one of the four components that can be contained in a DBE. Also, the second parameter must be a #define constant. There some constants that are universally valid and can be used with every DBE and other constants that may be used only with a specific DBE. The difference can be recognized by the prefix of the #define constant. The prefix DBE_ identifies universally valid constants and the prefix DBFDBE_ identifies constants which are only valid for the DBFDBE (more precisely, for the DATA component which manages DBF files). Constants containing the prefix DBFDBE_ are defined in the #include file DBFDBE.CH. The third parameter specifies the value which is to be set for the specific setting (in the example, the file extension) of a DBE.

Not all settings of a DBE are changeable, so the third parameter is only processed by DbeInfo() when the DBE allows the corresponding setting to be changed. The following table gives an overview of the universally valid settings that exist for all DBEs. The only setting which can be changed is the file extension:

Universal constants for characteristics of DatabaseEngines

Constant	*)	Data type	Description
DBE_DATATYPES	ro	C	Supported data types
DBE_EXTENSION	a	C	Default file extension
DBE_MANUFACTURER	ro	C	Producer of the DBE
DBE_NAME	ro	C	Name of the DBE
DBE_VERSION	ro	C	Version of the DBE

**) ro=READONLY , a=ASSIGNABLE*

Along with these general constants, most DatabaseEngines have specific #define constants that can only be used for the specific DBE (more precisely, for a specific component). In the section "Specifications of the DatabaseEngines", DBE specific constants for DbeInfo() are described.

12.3.2. The function DbInfo()

When a file is opened in a work area, a database object (DBO) is created by the current DatabaseEngine to manage the file open in the work area. The DBO represents the work area and the function DbInfo() can read information about the DBO and can change settings of the DBO. DbInfo() requires that a file be open in the corresponding work area.

DbInfo(), as well as DbeInfo(), receives parameters that are constants defined in an #include file. Universally valid constants can be found in the file DMLB.CH and are listed in the next table:

Universal constants for database objects (work areas)

Constant	*)	Data type	Description
DBO_ALIAS	ro	C	Alias name
DBO_FILENAME	ro	C	Name of the open file
DBO_ORDERS	ro	N	Number of orders (indexes)
DBO_RELATIONS	ro	C	Number of relations
DBO_SHARED	ro	L	.T. if the file is opened in SHARED mode
DBO_REMOTE	ro	L	.T. if the file is not stored on the local workstation
DBO_DBENAME	ro	L	Name of the DatabaseEngine that has opened a database file

**) ro=READONLY , a=ASSIGNABLE*

The universally valid #define constants for DbInfo() starts with the prefix DBO_ (for database object). There are also constants which can be used only with DBOs created by a specific DatabaseEngine. An example of constants that can be passed to DbInfo() is given in the following program code:

```
#include "Dmlb.ch"
#include "DbfDbe.ch"

DbeLoad( "DBFDBE", .T.)
DbeLoad( "NTXDBE", .T.)                  // create DBFNTX
DbeBuild( "DBFNTX", "DBFDBE", "NTXDBE" ) // compound DBE

USE Customer ALIAS Cust

? DbInfo( DBO_ALIAS )                // result: Cust
? DbInfo( DBO_FILENAME )             // result: C:\DATA\Customer.DBF

                                     // file handles
? DbInfo( DBFDBO_DBFHANDLE )         // result: 8
? DbInfo( DBFDBO_DBTHANDLE )         // result: 9
```

The example illustrates that the first parameter passed to DbInfo() is a #define constant which is either a universally valid constant (prefix DBO_) or a specific DBE constant. In the case of the DBFDBE, the specific constants for the function DbInfo() start with the prefix DBFDBO_ and are contained in the #include file DBFDBE. (To summarize: constants which contain DBE_ are valid for a DatabaseEngine, and for the function DbeInfo(). Constants which contain DBO_ are valid for database objects and for the function DbInfo()). The constants for DbInfo() that are dependent on the current DatabaseEngine are listed in the section "Specifications of the DatabaseEngines".

A DBO is initialized with all the current settings of the DBE when the file is opened. If changes are later made to the DBE using DbeInfo(), all DBOs remain unaffected by the change. This means that all work areas where files are open are not affected by changes made to a database engine. Such changes affect only those work areas where a file is opened after the change is made.

12.3.3. The function FieldInfo()

As soon as a file is opened in a work area, field variables (fields) exist within this work area. Similar to Clipper, information about a field can be determined using the functions FieldName() or FieldPos(). Xbase^{++} also includes the function FieldInfo() to read or change information about an individual field in a work area. The function FieldInfo() behaves in a manner similar to DbeInfo() and DbInfo(), and takes a #define constant as the second parameter. The valid constants for FieldInfo() are listed in the following table:

Universal constants for field variables in a work area

Constant	*)	Data type	Description
FLD_LEN	ro	N	Length of field
FLD_DEC	ro	N	Number of decimal places
FLD_TYPE	ro	N	Data type of field variable on the Xbase^{++} language level
FLD_NATIVETYPE	ro	N	Original data type of field variable as defined in the DBE

**) ro=READONLY , a=ASSIGNABLE*

FieldInfo() provides important pieces of information about fields in the database such as the length and the number of decimal places.

Example: (In this example, it is assumed that the DBFDBE is loaded)

```
#include "Dmlb.ch"

                                          // create database
DbCreate( "Part"     { { "PartNo"     , "C",  6, 0 }, ;
                       { "Part"       , "C", 20, 0 }, ;
                       { "Price"      , "N",  8, 2 }  } )

USE Part

? FieldInfo( 3, FLD_LEN )                 // result: 8
? FieldInfo( 3, FLD_DEC )                 // result: 2

? FieldInfo( 1, FLD_LEN )                 // result: 6
? FieldInfo( 2, FLD_LEN )                 // result: 20
```

The first parameter of the function FieldInfo() is the ordinal position of a field (as returned by FieldPos()) and the second parameter is a #define constant designating what information is being requested. To determine the length of a field or its decimal places, two simple pseudo functions can be defined for translation by the preprocessor into calls to FieldInfo():

```
#xtranslate  FieldLen( <nPos> ) => FieldInfo( <nPos>, FLD_LEN )
#xtranslate  FieldDec( <nPos> ) => FieldInfo( <nPos>, FLD_DEC )
```

The data type of a field is also important information and can be determined by passing the constant FLD_TYPE or FLD_NATIVETYPE. In both cases FieldInfo() returns a numeric value identifying the data type. Using the two constants, the data type which is available to be manipulated by the appropriate Xbase++ commands and functions can be distinguished from the original data type stored in the database. They are often, but not always identical. For example, at the language level of Xbase++ only a single numeric type exists. When numbers are stored in fields, however, integers and floating point numbers might be treated differently. Xbase++ recognizes the different representations for numbers and other data internally and distinguishes between data types that can exist on the language level and on the database level. Correspondingly, FieldInfo() can read the data type of a field as it exists on the language level (FLD_TYPE) or on the database level (FLD_NATIVETYPE). To determine the data type of a field on the Xbase++ language level, the constant FLD_TYPE is passed to FieldInfo(). The data type is returned by FieldInfo() as a numeric value, rather than by a character value such as that returned by Valtype() or Type(). The numeric identification of data types uses constants defined in the #include file TYPES.CH. The constants from the following table are available for determining data types using FieldInfo()

Constants for data types (FieldInfo(FLD_TYPE) return values)

Constant	Description
XPP_CHARACTER	Character value
XPP_DATE	Date value
XPP_LOGICAL	Logical value
XPP_MEMO	Memo field
XPP_NUMERIC	Numeric value
XPP_ARRAY	Array *)
XPP_BLOCK	Code block *)
XPP_DOUBLE	Numeric value as double*)
XPP_ILLEGAL	Invalid data type*)
XPP_OBJECT	Object *)
XPP_UNDEF	Undefined value (NIL)*)

**) Return values with FieldInfo() are dependent on the DBE*

12.4. Specifications of the DatabaseEngines - file formats

This section describes the specifications of the DatabaseEngines which are delivered with Xbase^{++}. It also gives information about database operations that are not supported by each of the DBEs.

12.4.1. SDFDBE (DATA component)

The SDFDBE manages ASCII files in System Data Format. The default file extension is ".TXT". Each line in the file has the same length and contains one record. Character, date, logical and numeric data types are supported, but the memo data type does not exist under the SDFDBE. The specification for a TXT file is as follows:

Specification for the System Data Format (TXT file)

Element	Specification
File extension	TXT *)
File size	Limited only by system resources
File end	Chr(26)
Record end	Carriage return + line feed = Chr(13)+Chr(10), all records have the same number of characters
Field separation characters	None

Element	**Specification**
Data types	C, D, L, N, no memo
Character values	Padded with blank spaces on the right
Date values	YYYYMMDD
Logical values	T or F *)
Numeric values	Left filled with zeros
Decimal character	Period *)

**) configurable*

When a TXT file in SDF format is created, either with COPY TO...SDF, DbExport(), or DbCreate(), a second file is created containing the structure description for the fields in the TXT file. By default this file has the extension SDF and is an ASCII file which could also be created using a simple text editor. The structure definition is supported in the familiar INI file format that can be written in its generalized form as follows:

```
[<Section>]
<Keyword>=<Value>
```

In an SDF file there are two sections: [INFO] and [FIELDS]. The following program example creates a TXT file in SDF format which illustrates this:

```
PROCEDURE Main
   LOCAL i

   DbeLoad( "SDFDBE" )                  // create TXT file
   DbCreate( "TEST", { {"CHARACTER", "C", 10, 0}, ;
                       {"DATE"     , "D",  8, 0}, ;
                       {"LOGICAL"  , "L",  1, 0}, ;
                       {"NUMERIC"  , "N",  6, 2}  }, ;
             "SDFDBE" )

   USE Test VIA SDFDBE                  // open using SDF engine

   FOR i:=1 TO 10                       // append 10 records
      DbAppend()
      REPLACE FIELD->character WITH Replicate( Chr(64+i), i ), ;
              FIELD->Date      WITH Date()+i                 , ;
              FIELD->Logical   WITH (i % 2 == 0)             , ;
              FIELD->Numeric   WITH (i ^ 2) / 2
   NEXT

   CLOSE Test                           // close file
RETURN
```

The example creates a TXT file with four fields and appends 10 records to the file. The file TEST.SDF is created as a structure file along with the data file TEST.TXT holding the data created in the FOR..NEXT loop. The structure file TEST.SDF looks like this:

```
[INFO]
file=TEST.TXT
fieldcount=4
recsize=27
reccount=10

[FIELDS]
CHARACTER=C,10,0
DATE=D,8,0
LOGICAL=L,1,0
NUMERIC=N,6,2
[END]
```

The first section [INFO] describes the file TEST.TXT that contains the data. This includes the file name without directory (file), the number of fields (fieldcount), the length of a record (recsize) and the number of records (reccount). The second section [FIELDS] describes the fields in the file TEST.TXT. Field names appear on the left side of the equals operator and data type, field length and decimal places are specified as a comma separated list on the right side. The definition is terminated with the section [END]. Note that both files (SDF and TXT file) must be located in the same directory.

After the example code above has run, the file TEST.TXT contains ten records with the following contents:

```
A         19950822F000.50
BB        19950823T002.00
CCC       19950824F004.50
DDDD      19950825T008.00
EEEEE     19950826F012.50
FFFFFF    19950827T018.00
GGGGGGG   19950828F024.50
HHHHHHHH  19950829T032.00
IIIIIIIII 19950830F040.50
JJJJJJJJJJ19950831T050.00
```

The SDF DatabaseEngine manages this file and allows database operations to be performed on the file. This includes defining filter conditions using SET FILTER or DbSetFilter(), as well as creating relations with DbSetRelation() or SET RELATION. If the SDFDBE is connected to a compound DBE that contains the NTXDBE (SDFNTX), indexes can also be created for an ASCII file existing in the SDF format.

The SDF format imposes certain restrictions in programming. For example, it is not possible to delete records, because there is no provision in the SDF file format for deletion flags. For this reason, a call to the function DbDelete() causes a runtime error if the file is managed using the SDFDBE. Also, the SDFDBE opens the file exclusively which means ASCII files in the SDF format can not be simultaneously used in the concurrent operation of several programs. The following table lists all common operations not supported by the SDFDBE:

Database operations that are not supported by the SDFDBE

Function/Command	Result of call
DbDelete()	Runtime error
DbRecall()	Runtime error
DbPack()	Runtime error
DbRLock()	Runtime error
DbSort()	Runtime error
RLock()	Runtime error
DbRLockList()	Returns an empty array
Deleted()	Always returns .F.
FLock()	Always returns .T.
Header()	Returns zero
USE...READONLY	READONLY is ignored
USE...SHARED	SHARED is ignored

Configuration of the SDFDBE

The SDF DatabaseEngine can be configured in specific ways using the function DbeInfo(). The SDF file format and the access mechanism can be affected when the SDF or TXT file are created. The following table gives an overview of the special #define constants that can be passed to the function DbeInfo() when the DatabaseEngine is SDFDBE:

Constants for DbeInfo() with the SDF-DBE

Constant	*)	Value	Data type	Description
SDFDBE_AUTOCREATION	a	.F.	L	TXT file is automatically created by DbCreate()
SDFDBE_DECIMAL_TOKEN	a	.	C	Character for decimal point
SDFDBE_LOGICAL_TOKEN	a	TF	C	Character for logical values
SDFDBE_STRUCTURE_EXT	a	SDF	C	Extension for the structure file

**) ro=READONLY, a=ASSIGNABLE*

The default values are shown in the column "Value".

SDFDBE_AUTOCREATION

This constant determines whether only the SDF file or the SDF file and the TXT file are created by DbCreate(). The default value for this setting is .F. (false) and the function DbCreate() creates only the SDF file (structure file) and not the TXT file by default. If the TXT file does not exist when USE or DbUseArea() opens the file, it is created.

If this setting is .T. (true), DbCreate() generates the SDF file as well as an empty TXT file. Any previously existing TXT file with this name is overwritten.

SDFDBE_DECIMAL_TOKEN

The delimiter for decimal places in numeric values can be set using this value. The default is the period. The comma is specified as the separator for decimal places in numeric values by the following code:

```
DbeInfo( COMPONENT_DATA, SDFDBE_DECIMAL_TOKEN, "," )
```

When the delimiter is specified as the ASCII character 0 (= Chr(0)) the SDFDBE ignores any delimiters for numeric fields. It then uses the field specification as found in the structure extended file (SDF file). Example:

SDF file:

```
CHAR=C,4,0
NUMERIC=N,6,2
```

TXT file:

```
AAAA004321
BBBB987654
```

In this case the field NUMERIC would have the value 43.21 for the first record and 9876.54 for the second one.

Note: This special operation mode of the SDFDBE has been introduced to provide for an easy data conversion between Xbase^{++} and VMS hosts.

SDFDBE_LOGICAL_TOKEN

This setting specifies the two characters used for logical values and specify the values that represent true and false. The default is "TF". A character string containing two characters is specified for this setting. The first character represents true and the second character represents false. The following program code specifies the character "1" for the logical value true and the character "0" for the logical value false.

```
DbeInfo( COMPONENT_DATA, SDFDBE_LOGICAL_TOKEN, "10" )
```

SDFDBE_STRUCTURE_EXT

This constant determines the file extension for the structure file (SDF file). The default is .SDF.

12.4.2. DELDBE (DATA Component)

The DatabaseEngine DELDBE manages ASCII files in delimited format. The default file extension is TXT. The delimited format is unique in that it allows records and the fields within records to have variable length. Fields are separated from each other by delimiters and the default field separator is the comma. Files in delimited format do not use a structure definition and there is no explicit typing or standardizing of fields.

An implicit typing is done when the fields are read based on the format in which various data is stored. Data of the character data type is enclosed in delimiting characters (double quotation marks by default). Numeric values are not enclosed in delimiting characters, and are recognized as numeric because they start with the digits 0 to 9, or a plus/minus prefix. Logical values are single alphabetical character which are not enclosed in delimiting characters. The default values are T and F. If two field separators occur next to each other without an alphanumeric character between them in a record, this is interpreted as the value NIL. The DELDBE is the only DBE delivered with Xbase^{++} capable of storing the value NIL. The data types date and memo are not supported by the DELDBE.

Specifications for the delimited format (TXT file)

Element	Specification
File extension	TXT *)
File size	Limited by system resources
File end	Chr(26)
Max. record length	Defaults to 128 Kilobyte *)
Record end	Carriage return + line feed (Chr(13) + Chr(10)) records can have a variable number of characters
Field separators	Comma *)
Data types	C, L, N, U, no date, no memo
Character values	Enclosed in double quotation marks *) blank spaces at the end are removed
Logical values	T or F, single alphabetical character *)
Numeric values	Digits 0 to 9, not zero filled
Decimal character	Period *)
NIL	Identified by two adjacent field separators with no characters in between

**) configurable*

The DELDBE manages ASCII files whose rows or records have variable lengths. This DBE is used with the commands COPY TO...DELIMITED and APPEND FROM...DELIMITED. It also supports basic database operations, like DbCreate(), DbUseArea(), and DbSkip(). Filters and relations can also be set with the DELDBE. Creating an index file is not supported, even if the DELDBE is coupled with the NTXDBE. The command DELETE and the function DbDelete() delete records but have a slightly different meaning. The deletion does not occur via a deletion flag (as with the DBFDBE), but results in an immediate physical deletion. After calling DbDelete(), a record can not be recalled when the file is managed by the DELDBE. The following table gives an overview of the database operations that are not supported by the DELDBE:

Unsupported database operations of the DELDBE

Function/Command	Result of call
DbRecall()	Runtime error
DbPack()	Runtime error
DbRLock()	Runtime error
DbSort()	Runtime error
DbSeek()	Runtime error
RLock()	Runtime error
FLock()	Runtime error
INDEX ON	Runtime error
SET INDEX	Runtime error
DbRLockList()	Returns an empty array
Deleted()	Always returns .F.
Header()	Returns 0
USE...READONLY	READONLY is ignored
USE...SHARED	SHARED is ignored

Field names and file structure (more precisely, the DbStruct() array) also behave in a special way under the DELDBE. An ASCII file in the delimited format contains no structure data, however fields of such a file can be accessed using an alias name. The field names begin with "FIELD" and are enumerated from 1 to FCount(). The following program code illustrates this:

```
USE Customer ALIAS Cust VIA DBFNTX              // open DBF file
aStructure := DbStruct()
                                                // create TXT file
DbCreate( "Address.txt", aStructure, "DELDBE" )
USE Address ALIAS Addr VIA DELDBE

Addr->( DbAppend() )                            // append record

REPLACE Addr->FIELD1 WITH Cust->LName, ;        // import data from DBF
        Addr->FIELD2 WITH Cust->FName, ;        // to TXT file
        Addr->FIELD3 WITH Cust->Phone
```

In this example a TXT file in delimited format is created using the function DbCreate(). The structure description is taken from a DBF file using the DbStruct() array. When a file is created using the DELDBE, the structure array passed to DbCreate() only determines the number of fields, not field names or field types. Since the length of fields is variable in the delimited format, the structure array can not set field lengths or the number of decimal places for numeric fields. Also calling DbStruct() for a file managed by the DELDBE provides a structure array which gives the structure of the current record only. It does not give the record structure of the file, since the fields can differ in structure from one record to another.

Configuration of the DELDBE with DbeInfo()

The DEL DatabaseEngine can be configured in specific ways using the function DbeInfo(). Specifically, the DEL file format and the operating mode of the DELDBE can be configured using DbeInfo(). The following table gives an overview of the special #define constants that can be passed to the function DbeInfo() for the DELDBE:

Constants for DbeInfo() with the DELDBE

Constant	*)	Value	Data type	Description
DELDBE_MODE	a	#define	N	Operating mode (see below)
DELDBE_RECORD_TOKEN	a	CRLF	C	Separation character for record
DELDBE_FIELD_TOKEN	a	,	C	Separation character for fields
DELDBE_DELIMITER_TOKEN	a	"	C	Embedding character for character values
DELDBE_DECIMAL_TOKEN	a	.	C	Character for decimal point
DELDBE_LOGICAL_TOKEN	a	TF	C	Characters for logical values
DELDBE_MAX_BUFFERSIZE	a	128	N	Max. size for record in KB

**) ro=READONLY, a=ASSIGNABLE*

The default values are listed in the column "Value".

DELDBE_MODE

This constant is used to set the operating mode of the DELDBE. There are three operating modes: auto-field, multi-field and single field. The default operating mode is auto-field. In the following line the auto field operating mode is changed to single field:

```
DbeInfo( COMPONENT_DATA, DELDBE_MODE, DELDBE_SINGLEFIELD )
```

The three possible operating modes are described in the following sections. The first line of the ASCII file in the delimited format differs depending on the operating mode.

Mode: DELDBE_AUTOFIELD

The DELDBE_AUTOFIELD mode is the default operating mode. The DELDBE is in this mode after it is loaded using DbeLoad(). If a TXT file is created using DbCreate() for a file in delimited format in auto-field mode, it contains a single line containing FCount()-1 commas. Thus, the first line of a delimited file in auto-field mode always contains commas representing the number of fields per record. Example:

```
#include "DelDbe.ch"

DbeLoad( "DELDBE" )                   // load DELDBE

aStruct := { {"CHAR1","C",10,0} , ;   // create structure array
             {"CHAR2","C",10,0} , ;   // for DbCreate()
             {"NUM"  ,"N", 6,2} , ;
             {"LOGIC","L", 1,0} }

DbCreate( "Auto", aStruct, "DELDBE" ) // create and open TXT
USE Auto VIA DELDBE                   // file
FOR i:=1 TO 3                         // append three records
   DbAppend()
   REPLACE FIELD->FIELD1 WITH Replicate(Chr(64+i),i), ;
           FIELD->FIELD2 WITH Replicate(Chr(96+i),i), ;
           FIELD->FIELD3 WITH 10^i, ;
           FIELD->FIELD4 WITH (i%2 == 1)
NEXT
```

In this code, a TXT file with four fields is created in delimited format. The fields do not correspond to the names from the structure array, but are numbered and are prefixed with "FIELD". After the above code runs, the file "Auto.txt" contains:

```
,,,
"A","a",10.00,T
"BB","bb",100.00,F
"CCC","ccc",1000.00,T
```

In the first line there are three commas (more precisely, field separators), which correspond to the number of fields minus one. The lines that follow contain the data in delimited format: character values are enclosed in quotation marks, numeric values consist only of digits, and logical values consist of one alphabetical character.

Mode: DELDBE_MULTIFIELD

The DELDBE_MULTIFIELD mode is an operating mode that was created especially for control files for form letters. Control files are used by text processing programs and contain the variable data for form letters. Generally the first line of a control file contains the field names, or variable names defined in the form letter that are replaced by values from the control file. In multi-field operating mode, the first line of the TXT file contains the field names. The following example creates a typical control file where the delimiting characters for character values and the separators for fields are changed:

```
#include "DelDbe.ch"

? DbeLoad( "DELDBE"" )
? DbeSetDefault( "DELDBE" )

aStruct := { {"CHAR1","C",10,0} , ;
             {"CHAR2","C",10,0} , ;
             {"NUM"  ,"N", 6,2} , ;
             {"LOGIC","L", 1,0} }
                                      // switch to multi-field
DbeInfo( COMPONENT_DATA, DELDBE_MODE, DELDBE_MULTIFIELD )
                                      // semicolon instead of
                                      // comma
DbeInfo( COMPONENT_DATA, DELDBE_FIELD_TOKEN, ";" )
                                      // no separator for
                                      // character values
DbeInfo( COMPONENT_DATA, DELDBE_DELIMITER_TOKEN, Chr(0) )

DbCreate( "Multi", aStruct, "DELDBE" )
USE Multi VIA DELDBE

FOR i:=1 TO 3                         // fields have field names
   DbAppend()
   REPLACE FIELD->CHAR1 WITH Replicate(Chr(64+i),i), ;
           FIELD->CHAR2 WITH Replicate(Chr(96+i),i), ;
           FIELD->NUM   WITH 10^i, ;
           FIELD->LOGIC WITH (i%2 == 1)
NEXT
```

In this code, a TXT file in delimited format is created containing four fields. The field names correspond to the names from the structure array and are stored in the first line of the TXT file. A semicolon is used as the field separator and character values are not enclosed in delimiting characters. After this example runs, the file "Multi.txt" contains:

```
CHAR1;CHAR2;NUM;LOGIC
A;a;10.00;T
BB;bb;100.00;F
CCC;ccc;1000.00;T
```

The first line contains the field names which correspond to the variable names of the control file. Subsequent lines contain the data for the form letter. The default action of delimiting character values is turned off. This is done by specifying the character Chr(0) as the delimiting character. The semicolon is defined as the field separator.

In "multi-field" operating mode, fields of a TXT file in delimited format can be specified by their field names, since the field names are defined in the TXT file. This is a special case and is generally used only to create control files for text processing from DBF files.

Mode: DELDBE_SINGLEFIELD

The mode DELDBE_SINGLEFIELD is an operating mode that allows any ASCII file to be opened and managed by the DELDBE. In this mode, the DELDBE treats an ASCII file as if the file contained only a single field. The field is treated as containing no delimiters, but characters indicating the end of a record are required. For an ASCII file, this generally corresponds to end of line characters (carriage return+line feed, Chr(13)+Chr(10)). In "single field" mode, a field is identical to a record and the DELDBE reads an ASCII file line by line. This allows database operations like DbSkip(), DbGoTop() or DbGoBottom() to be executed on an ASCII file. In this operating mode the DELDBE can manage many different ASCII files. It should be noted that in this mode the first line of an ASCII file is interpreted as "data" and not read for field information. Example:

```
#include "DelDbe.ch"

? DbeLoad( "DELDBE" )
? DbeSetDefault( "DELDBE" )

aStruct := { {"CHAR1","C",10,0} , ;
             {"CHAR2","C",10,0} , ;
             {"NUM"  ,"N", 6,2} , ;
             {"LOGIC","L", 1,0} }
                                    // switch to single field
DbeInfo( COMPONENT_DATA, DELDBE_MODE, DELDBE_SINGLEFIELD )

DbCreate( "Single", aStruct, "DELDBE" )
USE Single VIA DELDBE

FOR i:=1 TO 3
   DbAppend()
   REPLACE FIELD->FIELD1 WITH Replicate(Chr(64+i),i)
NEXT
```

Regardless of the structure array provided to DbCreate(), a TXT file in the single field mode is created as an empty file containing no field information. The DELDBE treats the TXT file as if it is a file with a single field. The field name is FIELD1. With exception of characters that represent the end of a record, the DELDBE does not recognize any other delimiting characters in this operating mode. The contents of the file "Single.txt" created in the above example would be as follows:

```
A
BB
CCC
```

Each line of the file contains only the data written into the file using REPLACE after DbAppend() added a new record.

The "single field" operating mode is suitable for viewing any ASCII file with rows that can be identified by distinct end of line character(s).

DELDBE_RECORD_TOKEN

The end of a record in a TXT file is generally identified by the two characters carriage return+line feed (Chr(13)+Chr(10). The delimiter for a record can be changed using the function DbeInfo(). The delimiter must contain at least one character and can have a maximum of two characters.

DELDBE_FIELD_TOKEN

In the delimited format (auto-field and multi-field mode), fields are set off from each other by a separator. The comma is the default separator. The following program line specifies the semicolon as the separator:

```
DbeInfo( COMPONENT_DATA, DELDBE_FIELD_TOKEN, ";" )
```

DELDBE_DELIMITER_TOKEN

Values of the "character" type are bracketed by certain characters in the delimited format. The default is the double quotation mark. The following call specifies the single quotation mark as the delimiting character for character values:

```
DbeInfo( COMPONENT_DATA, DELDBE_DELIMITER_TOKEN, "'" )
```

DELDBE_DECIMAL_TOKEN

The character used to mark decimal places in numeric values can also be specified. The default is the period. The following program line specifies the comma as the separator for decimal places in numeric values:

```
DbeInfo( COMPONENT_DATA, DELDBE_DECIMAL_TOKEN, "," )
```

When the character for decimal places is changed, it must not be the same as the field separator. If it is the same, places after the comma are looked at as a separate field containing another numeric value.

DELDBE_LOGICAL_TOKEN

For logical values, the two characters representing true and false can be specified. The default is "TF". If this setting is changed, a character string consisting of two characters must be specified. The first character represents true and the second character represents false. The following call specifies the character "Y" for the logical value true and the character "N" for the logical value false:

```
DbeInfo( COMPONENT_DATA, DELDBE_LOGICAL_TOKEN, "YN" )
```

Alphabetical characters must be indicated for logical values used with the DELDBE. The characters "1" and "0" for "true" and "false" are not permitted, since they are interpreted as numeric values.

DELDBE_MAX_BUFFERSIZE

The delimited format allows fields and records to become larger or longer. A character field containing 10 characters can be assigned a character string containing 20K characters. The memory space occupied by a field in a record must be dynamically enlarged (and also reduced) after an assignment. This automatically occurs with the DELDBE. The dynamic allocation of memory space in a file (not in memory) implies a considerable limitation of speed in database operations with the DELDBE. To optimize the speed of the DELDBE, the read buffer for a record loaded using the DELDBE is limited to 1 KB (1024 bytes). As soon as a record longer than 1024 byte is read, the read buffer is doubled. This behavior requires that an upper limit be defined, or the maximum length of a record which no record can exceed. The upper limit for the length of a record is set at 128KB by default. When any line of an ASCII file managed by the DELDBE is anticipated to be longer than 128KB characters, the maximum size of the read buffer must be increased. The following program line increases the maximum size of the read buffer of the DELDBE to 256 kilobytes:

```
DbeInfo( COMPONENT_DATA, DELDBE_MAX_BUFFERSIZE, 256 )
```

12.4.3. DBFDBE (DATA component)

The DatabaseEngine DBFDBE manages files in the DBF format. This file format is used by all Xbase dialects to store data. In a DBF file, data is stored in the form of a table where a record represents a row and a field represents a column. Rows in the table have a fixed length, and each field stores values which always have the same data type.

The DBFDBE is used as a DATA component of the compound DBE DBFNTX which is created each time an Xbase^{++} program starts. This occurs in the function DbeSys() whose source code is contained in the file ..\SOURCE\SYS\DBESYS.PRG. The DBFDBE performs all database operations that can be executed on a DBF file. This includes navigation using DbSkip() as well as the definition of filters and marking records for deletion using DbDelete(). In addition, the DBFDBE manages memo files which contain the text of memo fields.

Database operations that require an index can not be performed by the DBFDBE. For these operations, the DatabaseEngine must be combined with an ORDER component to create a compound DBE. Xbase^{++} provides the NTXDBE which manages index files (see the next section).

The DBFDBE has some limitations which are caused by the format of DBF files or by factors associated with the concurrent operation with Clipper applications.

The following table gives an overview:

Specifications for DBF files

Element	Specification
File size	Limited to the offset for record locks, default is 1 GB (10^9 bytes)
Max. number of fields	Not limited
Max. number of records	(Offset for record locks - Header() - 1) / RecSize()
Data types	C, D, L, N, M
Character values	Max. 64 KB
Date values	Fields always 8 bytes
Logical values	Fields always 1 bytes
Numeric values	Fields contain max. 19 bytes, including decimal point and negative signed prefix
Decimal character	Period
Memo	Fields always 10 bytes; contents of the fields (text) stored in memo files and the length of the text in memo fields is not limited
Size of a memo file	Limited only by system resources

Fields in a DBF file are strongly typed and have a fixed length. This is predefined in the file structure. The maximum value that can be stored in numeric fields depends on field length and decimal point. For negative values, a minus sign limits the smallest value additionally. Any attempt to store a numeric value too large or too small to fit into a database field results in a runtime error. Therefore it is necessary to consider the range of numeric data to be stored when numeric fields are specified.

In case of character fields, the field length limits the maximum number of characters that can be stored. If the number of characters in a string exceeds the field length, the character string is truncated and no runtime error is raised.

Using DBFDBE there is no limitation on the length of memo fields and memo files, as there is with the DOS-based 64 KB limit. Memo fields can store text longer than 64 KB characters, and memo files (DBT files) can become larger than 32 MB. However, these issues must still be considered when Xbase^{++} applications use the same database as DOS xBase applications.

The maximum file size for a DBF file depends on the offset used for record locks during concurrent operation (see the description of DBFDBE_LOCKOFFSET below). Under Clipper, the offset varies from version 5.01 to 5.2 and can be set to a matching value in

Xbase++. This allows Xbase++ to operate concurrently with existing Clipper applications in a heterogeneous network where some of the work stations run under DOS and others under a 32bit operating system. The maximum number of records is dependent on both the offset for records and on the length of records. Generally, the maximum number of records can be calculated as the offset for records locks divided by the length of a record.

The DBFDBE uses two reserved identifiers for field variables: _LOCK and _NULL. _LOCK is used for automatic record locking and _NULL is reserved for future purposes. If the DBFDBE is run in autolock mode, a DBF file must have the _LOCK field (see below).

Configuration of the DBFDBE with DbeInfo()

The DBF DatabaseEngine can be configured in specific ways using the function DbeInfo(). For example, the file extension for database files can be specified, or the operating mode for record locks can be set. The following table gives an overview of the #define constants that can be passed to the function DbeInfo() for the DBFDBE:

Constants for DbeInfo() with the DBFDBE

Constant	*)	Value	Data type	Description
DBE_EXTENSION	a	DBF	C	Extension for DBF file
DBFDBE_MEMOFILE_EXT	ro	DBT	C	Extension for memo file
DBFDBE_MEMOBLOCKSIZE	ro	512	N	Block size for memos
DBFDBE_LOCKOFFSET	a	2*10^9	N	Offset for record locks
DBFDBE_LOCKMODE	a	DBF_NOLOCK	N	Mode for record locks
DBFDBE_LOCKRETRY	a	3	N	Maximum number of new lock attempts with RLock()
DBFDBE_LOCKDELAY	a	25	N	Time interval for new lock attempts with
RLock()				(Unit is 1/100 seconds)

**) ro=READONLY , a=ASSIGNABLE*

The default values are shown in the column "Value".

DBE_EXTENSION

This constant is valid for all DBEs and defines the default file extension for files which are managed by the DBE. The default file extension DBF for the DBFDBE can be changed. This is especially useful in protecting data. For example, some users use the DIR command to search for all files with a DBF extension. Simply changing the extension is enough to keep these users from attempting to manipulate the file outside the application which uses it.

```
   DbeInfo( COMPONENT_DATA, DBE_EXTENSION, "FBD" )
```

This call to DbeInfo() sets the default file extension for DBF files to "FBD".

DBFDBE_MEMOFILE_EXT

This constant allows the default file extension for memo files (DBT files) to be determined. The default file extension is predetermined by the DBT file format and can not be changed.

DBFDBE_MEMOBLOCKSIZE

This constant returns the minimum block size used to store text in a memo file. The default block size for the DBFDBE is 512 byte. It can not be changed. Within a memo file, texts or character strings are stored in multiple blocks each of which is 512 bytes. When text that is only 100 bytes long is stored, it still occupies 512 bytes in the memo file.

DBFDBE_LOCKOFFSET

This constant returns or changes the offset for record locks. The default offset is 2 gigabytes. It can be modified in order to guarantee concurrent operation of Xbase^{++} applications with Clipper applications. The offset is 1 gigabyte under Clipper 5.01 and 2 gigabytes under Clipper 5.2 when the NTXLOCK2.OBJ file is linked into the application. Changing the offset for record locks should only be done with a detailed knowledge of the lock mechanism for records. Example:

```
   DbeInfo( COMPONENT_DATA, DBFDBE_LOCKOFFSET, 10^9 )
```

In this program line the offset for record locks handled by the DATA component of the compound DBE DBFNTX is set to 1 gigabyte.

Warning: If the DBFDBE is set as a DATA component of a compound DBE, the change to the offset must also be performed for the ORDER component. For example, in the compound DBE DBFNTX, the DBFDBE and the NTXDBE must have the same offset for record locks. The offset for the NTXDBE is set using the constant NTXDBE_LOCKOFFSET.

DBFDBE_LOCKMODE

The DBFDBE supports record locks to allow concurrent operation of database applications. The DBFDBE has two operating modes for locking records. One operating mode is compatible with Clipper and requires explicit record locking and unlocking to be performed when any records are updated. The other mode uses the extended locking mechanisms of Xbase++ which support automatic record locking.

Mode: DBFDBE_NOLOCK

The mode DBFDBE_NOLOCK is the default operating mode. The DBFDBE is in this operating mode after it is loaded using DbeLoad(). This is also the mode of the compound DBE DBFNTX available each time an Xbase^{++} program starts. In concurrent operation in this operating mode, a record lock must be explicitly set using RLock() or DbRLock() before a value in a field can be changed. If new values are assigned to fields or field variables during concurrent operation without a record lock being set, a runtime error occurs. This behavior is compatible with Clipper.

Mode: DBFDBE_AUTOLOCK

In the DBFDBE_AUTOLOCK mode, the DBFDBE automatically executes a record lock when a record or a field is changed during concurrent operation. The lock is automatically removed after the change has been written to the file. This mode is not compatible with the lock mechanism implemented in Clipper. This means that this operating mode can not be used in concurrent operation between Xbase^{++} and Clipper programs. The problem arises because the DBFDBE_AUTOLOCK mode uses the lock mechanisms from the operating system instead of logical record locks set using a virtual offset. The operating system permits locking a record directly.

The automatic record lock smoothly handles a potential conflict that can occur during concurrent operation: the problem of the "lost update". This problem can occur when a record is processed at the same time by two programs A and B. Program A reads a record into memory, edits the values of the fields and writes the changes back to the database. Program B reads the record at the same time as program A, changes the values and writes its own set of changes back into the database after program A has just written its data. This situation causes a "lost update" scenario, because the data written into the database by program A is overwritten by the data of program B and is therefore lost.

The problem of the "lost update" is solved in the DBFDBE_AUTOLOCK mode because only the first set of data is automatically written to the file when two programs process the same record at the same time. When program B reads a record at the same time as program A, only the data from the program that first writes its changes back to the database is generally valid.

In order for the DBFDBE to recognize and avoid the scenario of the "lost update", a database must have a field _LOCK of type "C" with the field length 4. It is used to store a counter in binary format. The DBFDBE_AUTOLOCK operating mode does not function without the field _LOCK. The DBFDBE uses the value of the field _LOCK to recognize this potential problem situation and avoid the "lost update" scenario.

Example:

When program A reads a record, the value of _LOCK might be 100. At the same time program B reads the same record. The value of _LOCK for program B is also 100. Program A changes the data and writes the changes back into the record. In performing this update, the DBFDBE locks the record, writes the values into the fields and increments the value in the field _LOCK. The DBFDBE then unlocks the record. The value of _LOCK is now 101. When program B attempts to update the record with its own changes, the operation fails because the value of _LOCK is still 100 for program B. It does not match the value of _LOCK in the database. Program B wants to change data that has already been changed by program A. The data from program A is retained and the attempt by program B to gain write access causes a recoverable runtime error.

Such a runtime error should always be handled using the control structure BEGIN SEQUENCE...ENDSEQUENCE and an error code block. In this case, it is the responsibility of the programmer to decide whether the first or last change should be valid in the "lost update" scenario. An example of how the problem of the "lost update" can be handled is shown in the following program lines:

```
bError  := ErrorBlock( {|e| Break(e)} )

USE Customer

cLName   := Customer->LName           // read fields into
cFName   := Customer->Fname           // memory variables

@ 10,10 SAY " Last Name" GET cLName   // edit memory
@ 11,10 SAY "First Name" GET cFName   // variables
READ

DO WHILE .T.
   BEGIN SEQUENCE
      REPLACE Customer->LName   WITH cLName , ;
              Customer->FName   WITH cFName
      COMMIT
   RECOVER
      i := Alert( "Data has been changed by another station", ;
                {"Do not save", "Save anyway"} )
      IF i==2
         LOOP
      ENDIF
   ENDSEQUENCE
   EXIT
ENDDO

ErrorBlock( bError )
```

In this example, the runtime error that occurs in the case of the "lost update" scenario is captured using RECOVER and the decision whether the changed data should still be stored is the responsibility of the user. If the "lost update" runtime error occurs, the field _LOCK is reread and the value in the data buffer (memory) matches the value in the database field. It is then sufficient to rewrite the modified values into the database. In the example, this is accomplished using the LOOP statement which is only reached if there is an error.

DBFDBE_LOCKRETRY

When the DBFDBE is in DBFDBE_AUTOLOCK operating mode, it automatically sets a record lock as soon as data is to be written into the database (DBF file). In concurrent operation there is no guarantee that a record lock can be achieved (the record may be locked by another user). When a record lock fails the DBFDBE tries to relock the record up to a specified maximum number of times. The maximum number of attempts that will be made to lock a record is set using DBFDBE_LOCKRETRY. The default value is 3.

DBFDBE_LOCKDELAY

In DBFDBE_AUTOLOCK operating mode, the DBFDBE automatically tries to lock a record before changes are written into the database. If in concurrent operation the record lock fails, the DBFDBE repeats the attempt to lock the record a maximum of DBFDBE_LOCKRETRY times. By default, the DBFDBE waits 25 hundredths of a second (1/4 second) before it makes another attempt to lock the record. DBFDBE_LOCKRETRY is used to set the duration of the delay between attempts to lock a record. The unit is 1/100 second.

Retrieve information with the function DbInfo()

When a DBF file is opened, an instance of the DBFDBE is created. The instance is a database object (DBO) and includes all settings of the DBFDBE. The DBF database object represents the work area in which the DBF file is opened and it manages the open DBF file. The function DbInfo() can be used to retrieve information about the DBF DBO. A #define constant which identifies what specific information is required must be passed to the function. The constants that can be passed to function DbInfo() for the DBFDBE are listed in the following table:

Constants for DbInfo() with the DBFDBE

Constant	*)	Value	Data type	Description
DBO_FILENAME	ro		C	Complete file name
DBFDBO_MEMOFILE_EXT	ro	DBT	C	Extension for memo file
DBFDBO_MEMOBLOCKSIZE	ro	512	N	Block size for memo fields
DBFDBO_LOCKMODE	ro	#define	N	Mode for record locks
DBFDBO_LOCKOFFSET	ro	2*10^9	N	Offset for record locks
DBFDBO_LOCKRETRY	ro	3	N	Number of lock attempts
DBFDBO_LOCKDELAY	ro	25	N	Time interval between lock attempts (unit: 1/100 second)
DBFDBO_DBFHANDLE	ro	0	N	File handle of DBF file
DBFDBO_DBTHANDLE	ro	0	N	File handle of DBT file

**) ro=READONLY , a=ASSIGNABLE*

The function DbInfo() returns the settings for the instance of the DatabaseEngine DBFDBE active in a work area. Instead of DBFDBE_, the #define constants for DbInfo() begin with

DBFDBO_. In the case of a database object, these settings can only be read and not be redefined (READONLY instead of ASSIGNABLE).

12.4.4.FOXDBE (DATA component)

The DatabaseEngine FOXDBE manages database files compatible with FoxPro. Their structure complies with the DBF format but they may contain additional field types other than *Character*, *Date*, *Logical Numeric* or *Memo*. FoxPro specific field types are automatically converted to Xbase++ compatible data types when an Xbase++ application accesses databases created by FoxPro.

The major difference to the DBFDBE is the file format for memo files which is managed more efficiently. Memo files of the FOXDBE have FPT as their default file extension, not DBT. With the FPT file format, it is possible to store any binary data in memo fields, while only text can be stored in memo fields using the DBT file format. In addition, the block size which represents the minimum space used for data stored in a memo field can be defined. In the DBT format the block size is 512 bytes while the FOXDBE supports block sizes in the 33 bytes to 64 kb range.

The FOXDBE accounts for FoxPro specific features of the DBF format plus other factors associated with the concurrent operation with FoxPro applications. The following table gives an overview:

Specifications for FoxPro compatible DBF files

Element	Specification
File size	Limited to 2^31 bytes (2 gigabytes)
Max. number of fields	255
Max. number of records	(2^31 - Header() - 1) / RecSize()
Data types for fields	
FoxPro	B, C, D, F, G, I, L, N, M, T, Y
Xbase++	F, C, D, N, O, I, L, N, M, T, Y, V, X
Memo file size	Limited to 2^31 bytes (2 gigabytes)
Memo block size	64 Bytes (adjustable between 33 bytes and 64 kb)

FoxPro supports up to eleven different data types for fields when creating databases. In the data definition language (DDL), each field type is identified by a letter. The Xbase++ DDL uses the same letters for most of the field types. However, in some cases they are different to FoxPro:

Mapping of FoxPro field types in the Xbase++ DDL

Description	Field type	Length	Field type	Valtype()
FoxPro			Xbase++	
Double	B	8	F	N
Character (text) *)	C	1-254	C	C
Character (binary)	C	1-254	X	C
Date	D	8	D	D
Float	F	1-20	N	N
Generic	G	4	O	M
Long signed integer	I	4	I	N
Logical	L	1	L	L
Memo (text) *)	M	4 or 10	M	M
Memo (binary)	M	4 or 10	V	M
Numeric	N	1-20	N	N
Time stamp	T	8	T	C
Currency	Y	8	Y	N

**) Data is converted according to SET CHARSET*

If a numeric field of the type *Double* is to be created in a new database file, its field type is specified with the letter *B* in FoxPro, while Xbase++ uses the letter *F* for declaring a field as *Double* in the call to the DbCreate() function. These fields store numeric values in binary format and always require 8 bytes in the database. In this way, they differ from numeric fields having the DDL type "N" where numeric values are stored as a sequence of digits and where the range is limited by the field length. However, both types of fields store numeric data and the function Valtype() returns the letter *N* in both cases. Therefore, one must distinguish between the definition of a field (DDL) and the data type stored in a field. The FOXDBE extends the data definition language for DBF files but does not add new data types used for operations at runtime of an Xbase++ program.

For character fields and memo fields, the FOXDBE distinguishes text from binary data. Text is subject to an automatic codepage conversion according to SET CHARSET, while binary data is read from or written to a database without this conversion.

Configuration of the FOXDBE with DbeInfo()

The DatabaseEngine can be configured in specific ways using the function DbeInfo(). For example, the file extension for database files can be specified, or the block size for memo fields can be set. The following table gives an overview of the #define constants which can be passed to the function DbeInfo() for the FOXDBE:

Constants for DbeInfo() with the FOXDBE

Constant	*)	Value	Data type	Description
DBE_EXTENSION	a	DBF	C	Extension for DBF file
FOXDBE_MEMOFILE_EXT	a	FPT	C	Extension for memo file
FOXDBE_MEMOBLOCKSIZE	a	64	N	Block size for memo fields
FOXDBE_MODE	a	#define	N	Mode for FoxPro 2.x or 3.x
FOXDBE_LOCKOFFSET	a	2*10^9	N	Offset for record locks
FOXDBE_LOCKRETRY	a	3	N	Maximum number of new lock attempts with RLock()
FOXDBE_LOCKDELAY	a	25	N	Time interval for new lock attempts with RLock() (Unit is 1/100 seconds)

**) ro=READONLY , a=ASSIGNABLE*

The default values are shown in the column "Value".

DBE_EXTENSION

This constant is valid for all DBEs and defines the default file extension for files which are managed by the DBE. The default file extension DBF for the FOXDBE can be changed.

```
DbeInfo( COMPONENT_DATA, DBE_EXTENSION, "FOX" )
```

This call to DbeInfo() sets the default file extension for DBF files to "FOX".

FOXDBE_MEMOFILE_EXT

This constant is used to determine or change the default file extension for memo files (FPT files).

FOXDBE_MEMOBLOCKSIZE

This constant returns or changes the minimum block size used to store data in a memo file. The default block size for the FOXDBE is 64 bytes. Within a memo file, texts or character strings are stored in multiple blocks each of which is 64 bytes. When text which is only 100 bytes long is stored, it still occupies 128 bytes in the memo file.

FOXDBE_MODE

The FOXDBE can manage database files in FoxPro 2.x or 3.x format and higher (Visual FoxPro format). The following lines show how to select the DBF format:

```
// FoxPro 2.x
DbeInfo( COMPONENT_DATA, FOXDBE_MODE, FOXDBE_MODE_OLD )
```

```
// Visual FoxPro (default)
DbeInfo( COMPONENT_DATA, FOXDBE_MODE, FOXDBE_MODE_VISUAL )
```

By default, the FOXDBE uses the Visual FoxPro file format.

Retrieve information with the function DbInfo()

When a DBF file is opened, an instance of the FOXDBE is created. The instance is a database object (DBO) and includes all settings of the FOXDBE. The DBF database object represents the work area in which the DBF file is opened and it manages the open DBF file. The function DbInfo() can be used to retrieve information about the DBO (the work area). A #define constant which identifies what specific information is required must be passed to the function. The constants that can be passed to function DbInfo() for the FOXBE are listed in the following table:

Constants for DbInfo() with the FOXDBE

Constant	*)	Value	Data type	Description
DBO_FILENAME	ro		C	Complete file name
FOXDBO_MEMOFILE_EXT	ro	FPT	C	Extension for memo file
FOXDBO_MEMOBLOCKSIZE	ro	64	N	Block size for memo fields
FOXDBO_MODE	ro	#define	N	FoxPro 2.x or 3.x database
FOXDBO_LOCKOFFSET	ro	2*10^9	N	Offset for record locks
FOXDBO_LOCKRETRY	ro	3	N	Number of lock attempts
FOXDBO_LOCKDELAY	ro	25	N	Time interval between lock attempts (unit: 1/100 second)

**) ro=READONLY , a=ASSIGNABLE*

The function DbInfo() returns the settings for the instance of the DatabaseEngine FOXDBE active in a work area. Instead of FOXDBE_, the #define constants for DbInfo() begin with FOXDBO_. In the case of a database object, these settings can only be read and not be redefined (READONLY instead of ASSIGNABLE).

12.4.5. NTXDBE (ORDER component)

The DatabaseEngine NTXDBE manages index files in the NTX format. This is the native Clipper index file format and used as the default format for building indexes in Xbase^{++}. The default can be changed by modifying the file DBESYS.PRG in the ..\SOURCE\SYS directory. The DbeSys() function in this file is automatically called at program start, prior to the Main procedure. A customized DBESYS.PRG must then be linked explicitly to the EXE file.

The NTX file format has undergone changes. The NTXDBE supports the NTX file format as it is defined by Clipper 5.2. An overview of its features is listed below:

Specifications for NTX files

Element	Specification
File size	Limited to the offset for record locks, the default is 2 GB (2*10^9 Byte)
Data types	C, D, L, N, no memo
Max. length for:	
- index expression	255 characters
- FOR expression	255 characters
Indexes per file	One
FOR expression	Supported
WHILE expression	Supported
TAG expression	Supported
EVAL...EVERY clause	Ignored

The only setting that can be set using DbeInfo() for the NTXDBE is the offset for record locks. Since record locks can be set in databases which are sorted by an index, the index file must also be locked. A record lock begins at a logical offset and the default offset is 2 gigabyte. This offset can be changed. When it is changed for the DBFDBE, the same value must be entered for the NTXDBE. An example for the compound DBE DBFNTX is:

```
DbeInfo( COMPONENT_ORDER, NTXDBE_LOCKOFFSET, 10^9 )
```

In this line of code, the offset for record locks managed by the ORDER component of the compound DBE DBFNTX is set to 1 gigabyte (compatible to Clipper 5.01).

12.4.6. CDXDBE (ORDER component)

The DatabaseEngine CDXDBE manages index files in the CDX format. This file format originated in FoxPro and has undergone changes. By default the CDXDBE supports the CDX file format as defined by FoxPro. In addition, the CDXDBE provides full interoperability with Clipper RDDs Six and Comix.

The major advantage of the CDX file format is its capability of maintaining multiple indexes in one index file. It also supports conditional indexes where only a subset of database records is referenced in the index. This is accomplished by a FOR condition which may be specified on index creation. An overview of the CDXDBE features is given in the following table:

Specifications for CDX files

Element	Specification
File size	Limited to the offset for record locks, the default is 2 GB (2 * 10^9 Byte)
Data types	C, D, L, N, no memo
Max. length for both index and FOR expression	512 characters
Indexes per file	Not limited
FOR expression	Supported
WHILE expression	Supported
TAG expression	Supported
EVAL...EVERY clause	Ignored

13. Multi-tasking and Multi-threading

This chapter describes functions which allow the Xbase^{++} programmer to access special capabilities of the operating system at the Xbase^{++} language level. These functions provide access to multi-tasking and multi-threading.

"Multi-tasking" is the capability of the operating system to run several application programs simultaneously. Each application program is embedded in a process. A process contains at least one "thread" where the program code is executed. Program code is executed only in a thread, not in a process. Several threads can be active within a process. This multi-threading capability of the operating system allows various parts of the same application program to run at the same time.

13.1. Start multiple processes - multi-tasking

The function RunShell() executes any program from within an Xbase^{++} application as a new process. The number of processes that can be started depends on available memory (including free space on the hard disk), but is limited by the operating system. To take full advantage of RunShell(), a comprehensive knowledge of the command START and of the commands associated with the command processor are required. When commands are passed to the command processor, the switch /C should always be considered. Detailed information is given in the online help of the operating system.

The function RunShell() starts a new instance of the command processor and passes a character string to it that is executed on the command line. Under Windows 95 the default command processor is COMMAND.COM and for OS/2 and Windows NT it is CMD.EXE. The function RunShell() must receive at least one parameter to be passed on to the command processor, which can be a null string (""):

```
xResult := RunShell( "" )
```

In the line above, RunShell() starts a new instance of the command processor, a new window is opened and the Xbase^{++} application is stopped until the window is closed. This is a result of the default values for other parameters that can be passed to RunShell(). These additional parameters can specify whether an Xbase^{++} application is dependent on or independent of the newly started command processor.

```
   // calling a program without specifying
   // command line parameters (""), but specifying
   // asynchronous execution (.T.)

   RunShell( "", "PROGRAM.EXE", .T.)
```

This line of code starts a program asynchronously in a new window. The program name is specified as the second parameter and no command line arguments are passed. The execution of the Xbase++ application which invoked RunShell() is continued by clicking on the Xbase++ application window with the mouse. When the third parameter contains the logical value .T. (true), the new program is started and the Xbase++ application continues to execute independently. If the value is .F. (false), the Xbase++ application waits until the newly started program has terminated before it continues executing.

The program can also be started as a background process and, in such case, the Xbase++ application remains active.

```
// Calling a program without specifying
// command line parameters ("").
// Use asynchronous execution (.T.)
// and run it as a background process (.T.)

RunShell( "", "PROGRAM.EXE", .T., .T.)
```

The fourth parameter determines whether the new process is started in the background or foreground of the Xbase++ application. By default, RunShell() starts new processes in the foreground. This means that the new window is brought to the front and receives input focus. If a "full screen" session is started using RunShell(), the function automatically switches to character mode. When the logical value .T. (true) is specified as the fourth parameter, the new process starts in the background of the Xbase++ application and the Xbase++ application keeps input focus.

The examples of calling a program show how an application can be started by passing the file name. The file name can also be passed to the function RunShell() as a command line parameter. The following two lines are equivalent:

```
RunShell( "", "PROGRAM.EXE", .T.)
RunShell("/C PROGRAM.EXE", , .T.)
```

In both cases, the program is started asynchronously. In the second line, the file name is contained in the command line parameter for the command processor. For this to work, the command line must begin with "/C". The following code shows additional examples for using the function RunShell():

```
// execute the program REVERSI.EXE asynchronously in
// the background

   RunShell( "", "REVERSI.EXE", .T., .T. )

// DOS session (/DOS) in the foreground (/F) in a window (/WIN)
// with the window title "DOS session" under OS/2

   RunShell( '/C START "DOS session" /F /WIN /DOS' )
```

```
// Full-Screen DOS session (/DOS) in the background (/B) under OS/2

   RunShell( '/C START /B /DOS' )

// Indirekt call of the Windows program EXPLORER.EXE
// using the START command.
// The explorer displays the current directory

   ? RunShell( "/C START EXPLORER.EXE ." )
```

The command START invokes a new command processor and passes a command line to it. This allows for the execution of any program including passing of parameters. When used with START, RunShell() offers tremendous flexibility in starting any program from an Xbase^{++} application and passing parameters to it (Note: enter "Help Start" or "Start /?" on the command line to get detailed information about the START command from the operating system's online help).

13.2. Using multiple threads

Multi-threading is a special characteristic of the operating system which allows an application program to be divided into various components which can be executed independently and simultaneously. The classic example for this is an application that provides the ability to evaluate data and print reports from one database while data input in another database is occurring. The idea is that the user starts one time-consuming procedure, and immediately begins working with another procedure while the first one is still running. In this example, the evaluation and reporting on one database runs in a different thread than the routine for data input. However, the database evaluation and report procedure, and the data input procedure are components of the same application. Another example is a program for data collection where the data is collected from various sources, with each source being controlled by a separate thread. Even when data is being received simultaneously from different sources, it can be reliably recorded.

13.2.1. Execution paths in a program

An application program (the EXE file) is started as a process. A process consists of one or more threads. Within a process, a thread can be thought of as a separate execution path where functions and procedures are executed independently from other threads. When a process consists of several threads, the operating system allocates the microprocessor (CPU) time for the different threads. Which thread gains access to the CPU (which thread is executed) depends first on the priority of a thread and then on whether it should execute instructions or whether it is currently in an idle mode. In multi-threading, the operating system allots each thread a limited amount of CPU time (called the time slice), and each thread is given its time in turn. In this way several processes are executed at the same time (multi-tasking), and within a process several threads can be executed (multi-threading). However, from the point of view of the CPU, only one thread is executing at any point in time.

The Thread class of Xbase++ offers the programmer a tool for taking advantage of multi-threading in a simple straightforward manner. A thread object must be created and the object must receive information as to what program code to execute in the thread. The following example shows the basic approach:

```
PROCEDURE Main
   LOCAL oThread := Thread():new()    // create thread object

   CLS
   oThread:start( "Sum", 10000 )      // sum numbers from 1
                                      // to 10000
   DO WHILE .T.                       // display characters during
      ?? "."                          // the calculation
      Sleep(10)
      IF ! oThread:active             // Check if thread still runs
         EXIT
      ENDIF
   ENDDO

   ? "The sum is:", oThread:result
RETURN

FUNCTION Sum( nNumber )
   LOCAL i, nSum := 0

   FOR i:=1 TO nNumber
      nSum += i
      IF i % 100 == 0                 // progress display
         DispOutAt( MaxRow(), 0, i )
      ENDIF
   NEXT
RETURN nSum
```

The program has the sole purpose of demonstrating the use of a thread object in a short example (otherwise it is meaningless). The thread object is created using the *:new()* method of the Thread class. As soon as a thread object is created, a new thread is available. The thread is then ready to run. The program code to be executed in the new thread is specified by calling the *:start()* method of the thread object. The first parameter is a character string specifying the identifier for the function or procedure to be executed in the thread. In the example, the user-defined function Sum() is specified. All other parameters (10000 in this case) are passed as arguments to the function being executed. After the method *:start()* is called, the specified program code is executed in the new thread.

In the example program, two loops run simultaneously. The DO WHILE loop in the Main procedure outputs a dot on the screen on each pass through the loop. The FOR...NEXT loop in the function Sum() simultaneously calculates a sum. When the FOR...NEXT loop in the new thread is terminated, the DO WHILE loop is also terminated, because the instance variable *:active* signals that the new thread is no longer executing code. The result of the calculation is contained in the *:result* instance variable of the thread object.

The thread is created when the thread object is created. Execution of code in the thread is started using the method *:start()*. The identifier for the function or procedure to be executed in the thread is passed to this method as a character string. Thc symbol or identifier for the function or procedure must be available at runtime. This means the function or procedure started in a new thread cannot be declared as a STATIC FUNCTION or STATIC PROCEDURE. As long as a thread is executing program code, the instance variable *:active* has the value .T. (true). When the thread has terminated, the return value of the last function or procedure executed in the new thread is assigned to the instance variable *:result*.

13.2.2. Visibility of variables in threads

The ability to divide a program up into different threads presents a new dimension for programmers who have not previously programmed in a multi-threading environment. Although creating threads is simple in Xbase^{++}, programming a multi-threaded application adds new complexity and requirements for the design of programs. In addition, new sources of error must be considered which may result from different parts of an application being executed at the same time. First of all, multi-threading affects the visibility of variables in different threads. The following table shows the differences:

Visibility of variables in different threads

Visibility	Storage class
Visible in the process (all threads)	PUBLIC STATIC
Visible in the thread (this Thread)	LOCAL PRIVATE FIELD

Variables declared as LOCAL or PRIVATE are only visible in the thread where the declaration occurred. The variables declared with PUBLIC or STATIC are visible in all threads of a process (application program). Field variables (FIELD) are visible in a work area of a work space. A work space is bound to a thread. Since work spaces can be moved between threads, field variables can become visible in different threads. At a given point in time, a field variable is visible only in one thread.

Whenever program code is divided up into different threads, the possibility of multiple threads having simultaneous access to the same variable (PUBLIC or STATIC) and changing it should be avoided. If multiple threads are modifying the same variable, the value of the variable is not predictable.

The following example demonstrates this situation:

```
STATIC snNumber := 0                    // file-wide visible STATIC

PROCEDURE Main
   LOCAL i, oThread := Thread():new()  // create thread object

   oThread:start( "Decrement" )         // decrement snNumber

   FOR i:=1 TO 10000
      snNumber ++                       // increment snNumber
   NEXT

   ? "Result:", snNumber                // theoretically this is 0
RETURN

PROCEDURE Decrement
   LOCAL i
   FOR i:=1 TO 10000
      snNumber --                       // decrement snNumber
   NEXT
RETURN
```

In the example, two FOR...NEXT loops run simultaneously in two different threads. The same STATIC variable is accessed in both threads. The first thread increments the variable 10000 times and the second thread decrements the variable 10000 times. Theoretically, the result would be the value zero. In practice this value is seldom reached. Generally the value of *snNumber* at the end of the program is greater than zero. This is because the operating system independently allocates the processor time for the two threads. The FOR...NEXT loop in the Main procedure begins incrementing as soon as the first thread is started. Switching between the threads takes time, and the STATIC variable increments several times before the second thread has actually started. When the FOR...NEXT loop in the first thread ends, the entire process is terminated, including the second thread. This means that the FOR...NEXT loop in the second thread is cancelled before the counter variable *i* reaches the value 10000. For this reason, the value of *snNumber* at the end of the program is almost always greater than zero.

This program demonstrates that programming multiple threads requires consideration of special issues. Simultaneous access to the same variables or files by multiple threads should be avoided. When two threads are using the same variables, the result (or the value of the variable) is not predictable, since the operating system allocates which thread is to receive available processor time. The thread which last performed an assignment sets the value of the variable. As a general rule, the part of the program that is to run in a separate thread should be written in such way that it can be compiled and linked as an independent program. All variables in a thread should be protected from access by other threads.

13.2.3. Priorities of threads

Multi-threading allows different programs to run at the same time or the same program code to be simultaneously executed multiple times in different threads. When a value is assigned to a variable, the value of the variable depends on the thread which is allocated processor time by the operating system. The processor time allocated to a thread can be influenced by its defined priority. So the result of the last example program can be changed if a single program line is added:

```
STATIC snNumber := 0                   // file-wide visible STATIC

PROCEDURE Main
   LOCAL i, oThread := Thread():new() // create thread object
                                       // increase priority level
   oThread:setPriority( PRIORITY_ABOVENORMAL )
   oThread:start( "Decrement" )        // decrement snNumber

   FOR i:=1 TO 10000
      snNumber ++                      // increment snNumber
   NEXT

   ? "Result:", snNumber               // this is always 0
RETURN

PROCEDURE Decrement
   LOCAL i
   FOR i:=1 TO 10000
      snNumber --                      // decrement snNumber
   NEXT
RETURN
```

In this example, the priority of the new thread is increased in relation to the current thread using the method *:setPriority()*. This causes processor time to be preferentially allocated to the new thread. This means that the FOR...NEXT loop in the Decrement procedure is processed first, since the thread in which this loop runs has a higher priority than the thread in which the Main procedure is running. In this case, the FOR...NEXT loop in the Decrement procedure runs before the FOR...NEXT loop in the Main procedure. The result of the program is always zero because ***snNumber*** is first decremented 10000 times and then incremented 10000 times.

The example represents an extreme case in which the order of execution can be precisely controlled by raising the priority of individual threads. Generally, the order of execution of threads (the allocation of processor time) depends on several factors which are controlled by the operating system.

By default, threads in an Xbase++ program have the priority PRIORITY_NORMAL and this is generally adequate. This results in an Xbase++ application being given processor time on equal precedence with most other programs. In normal situations, the priority should not be changed. Changing the priority requires a detailed knowledge of the manner in which the operating system distributes processor time. Threads receive processor time based on their priority. Low priority threads receive CPU access if no thread with higher priority is running or if a higher priority thread has entered a wait state. A higher priority may be temporarily assigned to a thread with a lower priority to allow it to be executed (starvation boost).

The thread object allows the priority of threads to be changed and it remains the programmer's responsibility to use this power responsibly. Raising the priority of threads only provides more processor time to the thread from the operating system. It does not cause the program to run faster. In the worst scenario, if the priority is set too high, multi-tasking and multi-threading are no longer possible, since the Xbase^{++} application (or a single thread in the Xbase^{++} application) is allocated all the processor time. In this case, other programs cannot run until the Xbase^{++} application has terminated. Changing the priority of threads demands special care. These settings directly influence preemptive execution of several programs, or processes, respectively (multi-tasking). They do not affect the performance of an individual Xbase^{++} application.

13.2.4. Getting information about threads

Two functions exist in Xbase^{++} which are very useful in the context of multi-threading. They are used in the implementation of program code where the thread object which executes this code is unknown. The functions are *ThreadID()* and *ThreadObject()*.

Each thread managed by a Thread object can be identified by a numeric ID. Thread IDs are consecutive numbers, i.e. the first thread has the ID 1 and it executes the Main procedure. A Thread object stores the thread ID in its instance variable *:threadID*. When the function ThreadID() is called, it retrieves the Thread object of the current thread and returns the value of the instance variable *:threadID*. This again is the numeric ID of the current thread.

The function *ThreadObject()* is used in a similar way. But instead of the numeric ID, it returns the complete Thread object which executes the function. Therefore, the result of the following expressions is always identical:

```
ThreadID()                 // Return value of a function
ThreadObject():threadID    // Value of an instance variable
```

13.2.5. Thread objects know the time

A thread is started by calling the *:start()* method of a Thread object. Normally, execution of program code within the thread begins immediately after the method is called. However, it is possible to define the exact time when the thread is to begin with program execution. This is done with the *:setStartTime()* method which must receive a numeric value indicating "seconds since midnight". Example:

```
oThread:setStartTime( 12*60*60 )   // 12 o'clock
oThread:start( {|| HighNoon() } )
```

A Thread object monitors the system timer. Therefore, the routine *HighNoon()* in the example is executed at 12 o' clock although the thread is started earlier. The current thread which has called the *:start()* method continues to run.

Another form of time-dependent execution of program code is provided by the *:setInterval()* method. It defines a time interval for repeated execution of program code by a Thread object. Each time the interval expires, the Thread object automatically restarts its thread. This functionality is also provided in a simplified form by the function SetTimerEvent(). A typical example for this is the continuous display of the current time which can be programmed in different ways:

```
// ------------------- Example 1 ---------------------------
SetTimerEvent( 100, {|| DispOutAt( 0, 0, Time() ) } )

// ------------------- Example 2 ---------------------------
oThread:start( "ShowTime_A" )

PROCEDURE ShowTime_A
   DO WHILE .T.
      DispOutAt( 0, 0, Time() )
      Sleep( 100 )
   ENDDO
RETURN

// ------------------- Example 3 ---------------------------
oThread:setInterval( 100 )
oThread:start( "ShowTime_B" )

PROCEDURE ShowTime_B
   DispOutAt( 0, 0, Time() )
RETURN
```

The result of all three examples is identical: the time is displayed once a second in the upper left corner of the screen (the unit for the time interval is 1/100ths of a second). The easiest implementation is given by the SetTimerEvent() function which repeatedly evaluates a code block.

A comparison of the procedures ShowTime_A() and ShowTime_B() reveals an important implication which results from using the method *:setInterval()*. Example #2 uses a DO WHILE loop and an explicit wait state (function Sleep()) for continuous display, while example #3 works continuously without a DO WHILE loop. In example #3, a time interval which is monitored by the Thread object is defined. Therefore, the procedure ShowTime_B() is executed each time the interval elapses, and the thread implicitly enters a wait state in between two execution cycles.

13.2.6. (De)Initialization routines for Threads

The program code invoked in a thread by calling the *:start()* method can be differentiated in greater detail by additional (de)initialization routines which are executed once at the beginning of a thread and once before it terminates. The instance variables *:atStart* and *:atEnd* of a Thread object serve this particular purpose. Both can be assigned names of functions or code blocks:

```
oThread:atStart := {|| DbUseArea( .T., , "CUSTOMER" ) }
oThread:atEnd   := {|| DbCloseArea() }
oThread:setInterval( 0 )
oThread:start( "CustomerList", {|| FIELD->CITY = "New York" } )

<other program code which runs parallel>


PROCEDURE CustomerList( bForCondition )
   IF Eval( bForCondition )
      QOut( FIELD->LASTNAME )
   ENDIF

   SKIP

   IF Eof()
      ThreadObject():setInterval( NIL )
   ENDIF
RETURN
```

In this example, a database query which lists data of all customers living in New York is programmed. The file is opened when the thread starts, i.e. before the query begins, and it is closed before the thread terminates. This occurs in the *:atStart* and *:atEnd* code blocks. The program code for the evaluation of the database is implemented without the typical DO WHILE .NOT. Eof() loop. This becomes possible because the time interval for repeated execution of this code is set to zero. As a result, the code is immediately started again whenever the RETURN statement is reached. When the record pointer is moved to the end of file (Eof() == .T.), the interval is set to NIL which causes the thread not to repeat the code but to terminate.

13.2.7. User-defined Thread classes

The Thread class can serve as superclass for user-defined Thread classes whose instances each have their own thread. Three methods are provided for use in derived Thread classes. They have the PROTECTED: visibility attribute and can therefore be used in subclasses only. These methods are *:atStart()*, *:execute()* and *:atEnd()*, of which at least the *:execute()* method must be programmed in a user-defined Thread class. It contains the code to be executed in the separate thread after the *:start()* method is called.

The example for the database query in the previous section is used as basis for the following Thread class which performs the same database operations:

```
oThread := CustomerList():new()
oThread:start( , {|| FIELD->CITY = "New York" } )

<other program code which runs parallel>

******************************
CLASS CustomerList FROM Thread
   PROTECTED:
   METHOD atStart, execute, atEnd
ENDCLASS

// Open database file when thread starts
METHOD CustomerList:atStart
   USE CUSTOMER
   ::setInterval( 0 )
RETURN self

// Perform database query
METHOD CustomerList:execute( bForCondition )
   IF Eval( bForCondition )
      QOut( FIELD->LASTNAME )
   ENDIF

   SKIP

   IF Eof()
      ::setInterval( NIL )
   ENDIF
RETURN self

// Close database file before thread terminates
METHOD CustomerList:atEnd
   CLOSE CUSTOMER
RETURN self
```

The user-defined class is instantiated and a code block with the condition for the query is passed to the *:start()* method. The three methods *:atStart()*, *:atEnd()* and *:execute()* are then automatically invoked and executed within the thread. The code block passed to *:start()* is also passed to the method *:execute()*. The code of this method is repeatedly executed until the end of file is reached.

The example shows the methods which can or must be implemented in a user-defined Thread class. It uses the mechanism for time-controlled repeated execution of code in a thread. Instead of *:setInterval(0 | NIL)*, a DO WHILE loop can be used as well.

13.2.8. Controlling threads using wait states

In multi-threaded programs, each thread can be viewed as a separate execution path in which different parts of a program may be executed at the same time. It is also possible to run one and the same part of a program simultaneously in multiple threads. A DO WHILE loop, for example, can be programmed once but may be executed 10 times at the same time. Therefore, all language elements which control program flow in one thread are not appropriate for controlling the program flow between multiple threads. This applies to statements like FOR..NEXT, DO WHILE..ENDDO, IF..ENDIF, DO CASE..ENDCASE or BEGIN SEQUENCE..ENDSEQUENCE. All of these control structures are translated by the compiler at compile time and are only valid for one thread.

The possibilities for coordinating different threads begin with halting the current thread until one or more other threads have terminated. The functions ThreadWait(), ThreadWaitAll() and the method *:synchronize()* of the Thread class are used for this purpose. Whenever one of these functions or the method is called, the current thread stops program execution and enters a wait state. The thread waits for the end of one or more other threads and resumes afterwards. While waiting, the thread consumes no CPU resoures. The following scheme demonstrates this:

```
Thread A              Thread B

running
   |
oThreadB:start()
   |                  running
   |                     |          // simultaneous execution
   |                     |          // of program code
oThreadB:synchronize(0)  |
                         |          // thread B executes code
wait state               |
                      RETURN        // thread B terminates
running                             // thread A resumes
   |
```

Thread A starts thread B and waits for its end at a particular point in the program by calling the *:synchronize()* method. It is not possible to terminate thread B explicitly from thread A.

Normally, the coordination of threads via wait states is necessary if one thread A needs the result of another thread B. For example, the calculation of extensive statistics can be done in multiple threads where each thread collects data from a particular database and calculates just one part of the statistic. The consolidation of the entire statistic then occurs in one single thread which needs to wait for the results of all other threads. In this scenario, all threads execute different parts of a program at the same time and must be coordinated or synchronized at a particular point in the program. The coordination can be implemented by one thread waiting for all others, or by one thread telling other threads to leave their wait state. The latter possibility requires a Signal object for communication between threads.

13.2.9. Controlling threads using signals

There is a possibility for coordinating threads which does not require a thread to terminate before another thread resumes. However, this requires the usage of an object of the Signal class. The Signal object must be visible in two threads at the same time. With the aid of a Signal object, one thread can tell one or more other threads to leave their wait state and to resume program execution:

```
Thread A              Thread B

running               running         // simultaneous execution
   |                     |            // of program code
   |                  oSignal:wait()  // thread B stops
   |
   |                  wait state      // thread A executes code
oSignal:signal()
   |                  running         // thread B resumes
   |                     |
```

Whenever a thread executes the *:wait()* method of a Signal object, it enters a wait state and stops program execution. The thread leaves its wait state only after another thread calls the *:signal()* method of the same Signal object. In this way a communication between threads is realized. One thread tells other threads to leave their wait state and to resume program execution.

13.2.10. Mutual exclusion of threads

As long as multiple threads execute different program code at the same time, the coordination of threads is possible using wait states as achieved with *:synchronize()* or ThreadWait(). However, wait states are not possible if the same program code is running simultaneously in multiple threads. A common example for this situation is adding/deleting elements to/from a globally visible array:

```
PUBLIC aQueue := {}

FOR i:=1 TO 10000                        // This loop cannot
   Add( "Test" )                         // be executed in
   Del()                                 // multiple threads
NEXT

**********************
FUNCTION Add( xValue )
RETURN AAdd( aQueue, xValue )

**************
FUNCTION Del()
   LOCAL xValue

   IF Len(aQueue) > 1
```

```
      xValue := aQueue[1]
      ADel ( aQueue, 1 )
      ASize( aQueue, Len(aQueue)-1 )
   ENDIF
RETURN xValue
```

In this example, the array *aQueue* is used to temporarily store arbitrary values. The values are retrieved from the array according to the FIFO principle (First In First Out). Function Add() adds an element to the end of the array, while function Del() reads the first element and shrinks the array by one element (note: this kind of data management is called a Queue).

When the functions Add() and Del() are executed in different threads, the PUBLIC array *aQueue* is accessed simultaneously by multiple threads. This leads to a critical situation in function Del() when the array has only one element. In this case, a runtime error can occur:

```
Thread A                Thread B

LOCAL xValue
IF Len(aQueue) > 1
   Thread is            Thread executes function completely
   interrupted by the
   operating system     LOCAL xValue
                        IF Len(aQueue) > 1
                           xValue := aQueue[1]
                           ADel ( aQueue, 1 )
                           ASize( aQueue, Len(aQueue)-1 )
                        ENDIF
                        RETURN xValue
   Thread resumes

   xValue := aQueue[1]

   Runtime error:
   Meanwhile, the array is empty
```

The operating system can interrupt a thread at any time in order to give another thread access to the CPU. If threads A and B execute function Del() at the same time, it is possible that thread A is interrupted immediately after the IF statement. Thread B may then run the function to completion before thread A is scheduled again for program execution. In this case, a runtime error can occur because the function Del() is not completely executed in one thread before another thread executes the same function.

The example function Del() represents those situations in multi-threading which require muliple operations to be executed in one particular thread before another thread may execute the same operations. This can be resolved when thread B is stopped while thread A executes the Del() function. Only after thread A has run this function to completion may thread B begin with executing the same function. Such a situation is called "mutual exclusion" because one thread excludes all other threads from executing the same code at the same time.

Mutual exclusion of threads is achieved in Xbase++ not on the PROCEDURE/FUNCTION level but with the aid of SYNC methods. The SYNC attribute for methods guarantees that the method code is executed by only one thread at any time. However, this is restricted to one and the same object. If one object is visible in multiple threads and the method is called simultaneously with that object, the execution of the method is serialized between the threads. In contrast, if two objects of the same class are used in two threads and the same method is called with both objects, the program code of the method runs parallel in both threads. As a result, mutual exclusion is only possible if two threads attempt to execute the same method with the same object. The object must be an instance of a user-defined class that implements SYNC methods. A SYNC method is executed entirely in one thread. All other threads are automatically stopped when they attempt to execute the same method with the same object.

The example with the beforementioned PUBLIC array *aQueue* must be programmed as a user-defined class in order to safely access the array from multiple threads:

```
PUBLIC oQueue := Queue():new()

FOR i:=1 TO 10000                    // This loop can run
   oQueue:add( "Test" )              // simultaneously in
   oQueue:del()                      // multiple threads
NEXT

***********
CLASS Queue                          // Class for managing
   PROTECTED:                        // a queue
   VAR aQueue

   EXPORTED:
   INLINE METHOD init
      ::aQueue := {}                 // Initialize array
   RETURN self

   SYNC METHOD add, del              // Synchronized methods
ENDCLASS

METHOD Queue:add( xValue )           // Add an element
RETURN AAdd( ::aQueue, xValue )

METHOD Queue:del                     // Remove first element
   LOCAL xValue                      // and shrink array

   IF Len(::aQueue) > 1
      xValue := aQueue[1]
      ADel ( ::aQueue, 1 )
      ASize( ::aQueue, Len(::aQueue)-1 )
   ENDIF
RETURN xValue
```

In this example, the Queue class is used for managing a dynamic array that may be accessed and changed from multiple threads simultaneously. The array is referenced in the instance variable *:aQueue* and it is accessed within the SYNC methods *:add()* and *:del()*. The Queue object which contains the array is globally visible. The execution of the *:del()* method is automatically serialized between multiple threads:

```
Thread A                   Thread B
   |                          |
 oQueue:del()                 |
   |                        oQueue:del()
 <...>
 ADel( ::aQueue, 1 )          thread is stopped
 <...>
 RETURN xValue                thread resumes
   |                        <...>
   |                        ADel( ::aQueue, 1 )
   |                        <...>
   |                        RETURN xValue
   |                          |
```

When thread B wants to execute the *:del()* method while this method is executed by thread A, thread B is stopped because it wants to execute the method with the same Queue object. Therefore, SYNC methods are used whenever multiple operations must be guaranteed to be executed in one thread before another thread executes the same operations. A SYNC method can be executed with the same object only in one thread at any time (Note: if a class method is declared with the SYNC attribute, its execution is serialized for all objects of the class).

14. User Interface and Dialog Concepts

A large part of the program code for most applications is associated with the user interface and allowing the user to access features of the application. The operating system offers two operating modes for the user interface, the VIO mode (Video Input Output Mode) and GUI mode (Graphic User Interface). The VIO mode is a text based operating mode which is selected when the full screen mode is active or when a DOS program is running in a DOS window. The GUI mode is a graphic operating mode. An example of the GUI mode is the desktop. Xbase^{++} supports both operating modes which makes it possible to easily port existing Xbase programs written for DOS to a 32bit operating system.

To simplify migration of existing DOS Xbase programs from the text based VIO mode to the GUI mode, Xbase^{++} also includes a special "hybrid" mode. In this mode, character based program elements can be mixed with graphic dialog elements. This allows the programmer to transition existing text based DOS Xbase programs in a stepwise manner to applications with a full graphic user interface.

This chapter describes how programs can be created for the different operating modes and illustrates various concepts for programming user interfaces. This includes aspects of data input and output for both the procedural programming of VIO applications and the event driven, object oriented programming of GUI applications.

14.1. Applications in character mode (VIO mode)

This section describes the most important commands, functions, and dialog concepts used in programming VIO applications. With only a few exceptions, all relevant language elements of Xbase^{++} are compatible with Clipper. Programmers familiar with Clipper can just read the "Keyboard and mouse" and "The default Get system" sections of this chapter. Note that the functionality of a VIO application is guaranteed in hybrid mode as well as in GUI mode.

14.1.1.Unformatted input and output

The simplest form of data input and output is unformatted. Data is input or output at the current position of the screen cursor or print head. Xbase^{++} provides a set of commands for unformatted input and output. These are listed in the following table:

Commands for unformatted input and output

Command	Description
? | ??	Output the result of one or more expressions
ACCEPT	Input characters at the current cursor position
DISPLAY	Output the contents of a database file
INPUT	Input an expression at the current cursor position
LIST	Output the contents of a database file
SET ALTERNATE	Turn output to a file on or off
SET COLOR	Set the screen color
SET CONSOLE	Turn screen output on or off
SET PRINTER	Turn printer output on or off
TEXT...ENDTEXT	Output one or more lines of text
TYPE	Output the contents of any file
WAIT	Input a single character

The three commands ACCEPT, INPUT and WAIT provide unformatted input. WAIT accepts a single keystroke while ACCEPT and INPUT allow any number of characters to be entered. Input via ACCEPT and INPUT is ended when the user presses the Enter key. Characters entered using INPUT are considered as an expression and are compiled using the macro operator (an error in the expression leads to a runtime error). Characters entered using the ACCEPT command remain unchanged and can be assigned to a memory variable as a character string.

The most commonly used commands for unformatted output are the single and the double question marks (? or ??). These are equivalent to the functions QOut() and QQOut(), respectively. The results of one or more expressions can be output using these commands. The default output device is the screen. The results of the expressions can also be saved in a file (after SET ALTERNATE ON) or sent to a printer (after SET PRINTER ON). If the command SET CONSOLE OFF is called before output, screen output is suppressed. After screen output has been suppressed, the screen output must be reactivated using SET CONSOLE ON after output to the file and/or printer is complete.

The commands LIST and DISPLAY are both used to output records from a database file. The command TYPE outputs the contents of any text file. The options TO PRINTER and TO FILE are valid with all three of these commands, so simultaneous output to a printer or a file can be performed without calling SET PRINTER ON or SET ALTERNATE ON. The screen output of these commands can be suppressed by first calling SET CONSOLE OFF.

The command SET COLOR changes the color for the display of screen output. The command is not really an output command, but allows the color of the output to be modified.

Detailed descriptions of the commands for unformatted input and output, including program examples, are found in the reference documentation.

14.1.2. Formatted input and output

Formatted input and output allows the exact position on the screen or printer (the row and column) for data input or output to be specified. Xbase^{++} includes commands and functions for formatted input and output. The commands are translated by the preprocessor to the equivalent function, which means that the difference between commands and functions is just a difference in syntax. The command syntax sometimes allow more readable program code, since many commands imply several function calls and the command syntax provides additional keywords. The following table lists the most important functions and commands for formatted input and output:

Commands and functions for formatted input and output

Command/function	Description
@	Position screen cursor and delete screen line
@...BOX	Output box on the screen
@...CLEAR	Delete screen area
@...GET	Input data
@...SAY	Output data
@...TO	Output box on the screen
CLEAR	Delete entire screen
Col()	Return column position of screen cursor
DispOut()	Output the result of an expression, and update cursor position
DispOutAt()	Like DispOut() but cursor position not updated
DevOut()	Output expression results on the current output device
Row()	Return row position of screen cursor
MaxCol()	Return maximum number of columns on the screen
MaxRow()	Return maximum number of rows on the screen
PCol()	Return current column position of the print head
PRow()	Return current row position of the print head
SaveScreen()	Save screen area
SET DEVICE	Specify current output device
SetColor()	Set or return screen color
SetPos()	Change screen cursor position
SetPrc()	Set row and column coordinates of the print head
RestScreen()	Redisplay a saved screen area

Some of these functions and commands affect only the position of the screen cursor but are included because the cursor coordinates mark the position where data is displayed. Other functions and commands manage the screen itself. In VIO mode, the origin (0, 0) of the coordinate system is the upper left corner of the screen or window. The lower right corner

(MaxRow(), MaxCol()) represents the largest coordinate values that are visible. The position of the cursor is set by specifying the Row() and Col() to either the command @ or the function SetPos(). The screen is generally cleared using CLEAR at the start of each program prior to displaying anything for the first time. The commands @...BOX and @...TO draw boxes on the screen. They are only valid for the screen and are not available for the printer.

The most important command for formatted output is @...SAY which outputs the result of an expression. Output using @...SAY can occur on the screen or on the printer. Formatted output differs from unformatted output in that simultaneous output on the screen and printer is not possible. Selecting an output device is done using the command SET DEVICE (TO PRINTER or TO SCREEN). Output on the printer is at the current position of the print head which can be set using the functions PRow() and PCol(). The function SetPrc() resets the internal values for the row and column coordinates of the print head but does not reposition the print head.

The command @...SAY can be expanded to include data input using the GET option. Alternatively, an input field can be defined using the command @...GET. These commands are used to define one or more data entry fields prior to the actual data input which occurs within the READ command or the function ReadModal(). READ and ReadModal() both activate the default Get system of Xbase^{++} (see the later section on this).

14.1.3. Keyboard and mouse

Under an operating system with graphic user interface, the mouse (not the keyboard) is the most important input device for controlling the application. Because of this, running programs or individual modules is not controlled by program logic but by "events". These events are usually caused by actions of the user. Events, which come from outside the application, are temporarily stored by the operating system in an event queue and then sequentially processed by the application. Some examples of events would be: the mouse was moved, the right mouse button was clicked, the left mouse button was double clicked, etc. A keypress is also an event, which shows that events can come from various devices.

Each event is identified within the program by a numeric code. Each key has an associated unique numeric value and different mouse events have different numeric codes. In Xbase^{++}, events are all handled by a group of event functions, regardless of their origin. There is also a group of functions which assure language compatibility with Clipper. These compatibility functions can only respond to the keyboard and they only consider the keyboard codes defined in Clipper. It should be noted that the numeric values associated with the various keys in Clipper do not necessarily match the event codes of the operating system or Xbase^{++} and the old keycodes are offered only for compatibility.
A correlation is found between key codes and ASCII characters only in the range of 0 to 255. These compatibility functions only consider the key codes defined in Clipper. The distinction between "event functions" and "keyboard functions" is also shown in the following table:

Event functions and keyboard functions

Function	Description
Event functions supporting both keyboard and mouse input	
AppEvent()	Read event and remove it from the queue
LastAppEvent()	Return the last event
NextAppEvent()	Read next event without removing it from queue
PostAppEvent()	Put event into the queue
SetAppEvent()	Associate event with a code block
SetMouse()	Toggle availability of mouse events on or off
Compatibility functions supporting only keyboard input	
Inkey()	Read key code
KEYBOARD	Write characters into keyboard buffer *)
LastKey()	Return last key code
NextKey()	Read next key code
SetKey()	Associate key code with a code block

) KEYBOARD is a command, not a function

A detailed description of the compatibility functions can be found in the reference documentation.

AppEvent() reads events from the queue and also removes them from the queue. The return value of AppEvent() is the numeric code that uniquely identifies the event. The function SetMouse(.T.) must have been previously called for the AppEvent() function to register mouse events in VIO mode. The following program example is terminated when the right mouse button is pressed:

```
#include "Appevent.ch"

PROCEDURE Main
   LOCAL nEvent := 0

   CLEAR
   @ 0,0 SAY "Press right mouse button to terminate"
   SetMouse(.T.)                        // register mouse events

   DO WHILE nEvent <> xbeM_RbDown       // event: right mouse button
      nEvent := AppEvent(,,,0)          // wait until event occurs

      IF nEvent < xbeB_Event            // event: keypress
         ? "The event code for the key is:", nEvent
      ELSE
```

```
         ? "The event code for the mouse is:" , nEvent
      ENDIF
   ENDDO

RETURN
```

The large number of possible events makes it impractical to directly program events using the numeric codes. Instead, the constants defined in the #include file APPEVENT.CH should be used. These constants all begin with the prefix xbe (which stands for **x**base **e**vent) followed by an uppercase letter or an underscore. The uppercase letter identifies the category for the event and the underscore separates the prefix from the rest of the descriptive event name:

Event categories

Category	Prefix	Example
No event	xbe_	xbe_None
Keyboard event	xbeK_	xbeK_RETURN
Base event	xbeB_	xbeB_Event
Mouse event	xbeM_	xbeM_LbDown
Xbase Part event	xbeP_	xbeP_Activate

In addition to the return value identifying the event, the function AppEvent() modifies two parameters that are passed by reference. These are called "message parameters" and contain additional information about the event. In an event driven system such as Xbase^{++}, it is often insufficient to receive only a single event code. For example, it is generally necessary to know the position of the mouse pointer if the event is "mouse click". When the function AppEvent() is called, two parameters must be passed by reference to contain additional information about the event after AppEvent() returns. If the event is a mouse click, the coordinates of the mouse pointer are contained in the first message parameter as an array of two elements. The following example illustrates this:

```
#include "Appevent.ch"

PROCEDURE Main
   LOCAL nEvent := 0, mp1, nRow, nCol

   CLEAR
   @ 0,0 SAY "Press right mouse button to cancel"
   SetMouse(.T.)                       // register mouse event

   DO WHILE nEvent <> xbeM_RbDown      // event: right mouse button
      nEvent := AppEvent(@mp1,,,0)     // wait until event
                                       // occurs
      IF nEvent < xbeB_Event           // event: keypress
         ? "Event code:", nEvent
      ELSEIF nEvent <> xbeM_Motion     // mouse events exept
```

```
                                         // 'mouse moved'
         nRow := mp1[1]                  // mouse coordinates
         nCol := mp1[2]
         @ nRow, nCol SAY "The event code is:" + Str(nEvent)
      ENDIF
   ENDDO

RETURN
```

The variable *mp1* is passed by reference to the function AppEvent(). After each mouse event it contains the row and column coordinates of the mouse pointer in an array of two elements. In the example, the contents of the array are assigned to the two variables *nRow* and *nCol.* The event code is output at this position, unless it is a mouse movement. The xbeM_Motion event occurs very frequently during mouse movement and would clutter the screen if displayed.

In VIO mode, the mouse coordinates are the most important information contained in the message parameters. In many other cases the parameters contain the value NIL in this operating mode but contain additional information when Xbase Parts are used (of course, Xbase Parts are not available in VIO mode) or when user-defined events are created using the function PostAppEvent(). The following example illustrates the basic relationship between the functions AppEvent() and PostAppEvent() which (along with their message parameters) create the basis for event driven programming:

```
#include "Appevent.ch"

#define  xbeU_DrawBox  xbeP_User + 1   // xbeP_User is the base value
#define  xbeU_Quit     xbeP_User + 2   // for User events

PROCEDURE Main
   LOCAL nEvent := 0, mp1, mp2

   CLEAR
   @ 0,0 SAY " Draw Box |  QUIT"       // mouse sensitive region
   SetMouse(.T.)                       // register mouse

   DO WHILE .T.                        // infinite event loop
      nEvent := AppEvent( @mp1, @mp2 ,,0 )

      DO CASE
      CASE nEvent == xbeU_DrawBox      // user event
         DrawBox( mp1, mp2 )           // mp1 = Date(), mp2 = Time()
                                       // from PostAppEvent()
      CASE nEvent == xbeU_Quit         // second user event
         QUIT

      CASE nEvent < xbeB_Event         // key was pressed
         @ MaxRow(), 0
```

```
            ?? "Key code is:", nEvent

         CASE nEvent == xbeM_LbClick      // left mouse button click
            @ MaxRow(), 0
            ?? "Coordinates are:", mp1[1], mp1[2]

            IF mp1[1]== 0 .AND. mp1[2] <= 21 // in mouse sensitive
                                             // region
               IF mp1[2] <= 15
                  PostAppEvent( xbeU_DrawBox, Date(), Time() )
               ELSE
                  PostAppEvent( xbeU_Quit )
               ENDIF
            ELSE
               @ mp1[1], mp1[2] SAY "No selection made"
            ENDIF

         ENDCASE
      ENDDO

   RETURN

   *********************************       // define position and display
   PROCEDURE DrawBox( dDate, cTime )       // box using mouse clicks
      LOCAL nEvent := 0, mp1, nTop, nLeft, nBottom, nRight

      SAVE SCREEN
      @ 0, 0 SAY "Click on upper left corner of box"

      DO WHILE nEvent <> xbeM_LbClick      // wait for left mouse click
         nEvent := AppEvent( @mp1,,, 0 )
      ENDDO

      nTop  := mp1[1]
      nLeft := mp1[2]

      nEvent := 0
      @ nTop, nLeft SAY "Click on lower right corner of box"

      DO WHILE nEvent <> xbeM_LbClick      // wait for left mouse click
         nEvent := AppEvent( @mp1,,, 0 )

         IF nEvent == xbeM_LbClick .AND. ;
            ( mp1[1] <= nTop .OR. mp1[2] <= nLeft )
            Tone(1000,1)                   // lower right corner
            nEvent := 0                    // is invalid
         ENDIF
      ENDDO
   nBottom  := mp1[1]
   nRight   := mp1[2]

   RESTORE SCREEN
   @ nTop, nLeft TO nBottom, nRight    // output box

   @ nTop+1,nLeft+1 SAY "Date:"        // display values of
   ?? dDate                            // PostAppEvent()
   @ nTop+2,nLeft+1 SAY " Time:"
   ?? cTime

RETURN
```

In the example, one of the two user-defined events is generated when the mouse is clicked in the hot spot region of the first screen row. The mouse coordinates (contained in the message parameter *mp1*) are used to distinguish whether the xbeU_DrawBox event or the xbeU_Quit event is placed in the queue by PostAppEvent(). The user-defined event xbeU_DrawBox receives the return values of Date() and Time() as message parameters. When this event is retrieved from the queue by AppEvent(), the message parameters are placed in the variables *mp1* and *mp2*. These values are passed to the procedure *DrawBox()* and displayed on the screen by this function.

The program is a simple example showing the logic of event driven programming. Events are identified by a unique numeric value (using a #define constant) and additional information about the event is contained in the two message parameters. The values contained in the two parameters *mp1* and *mp2* vary depending on the event. The message parameter values can range from NIL in the simplest case to complex data structures such as arrays or objects. PostAppEvent() can be used to place any event in the queue that can then be retrieved from any place in the program using AppEvent().

14.1.4. The default Get system

Xbase^{++} provides a Get system for formatted data input in VIO mode. The source code of this system is contained in the GETSYS.PRG file. The open architecture of the Get system offers tremendous possibilities such as data validation before, during and after data entry. It also offers a specific place where a programmer can identify and process events that occur during data input without having to change the basic language elements for defining input fields.

An input field is most easily created using the command @...SAY...GET. The data entry itself is started using the command READ:

```
USE Address ALIAS Addr

@ 10,10 SAY "First Name:" GET Addr->FIRSTNAME  // define data
@ 12,10 SAY " Last Name:" GET Addr->LASTNAME   // entry fields

                                               // read input
READ
```

In these four lines a database file is opened, two data entry fields are defined and input is performed via the READ command. Input is terminated when the entry in the second field is finished using the Return key or if the Esc key is pressed during input.

The command syntax is the easiest way to program data entry fields. This syntax is translated by the preprocessor into code that creates Get objects and stores them in the *GetList* array. Get objects include methods that allow the formatted display of values and interactive data entry. The data entry values can be contained in memory variables or in field variables. Access to the contents of data entry variables does not occur directly, but through a code block. This code block is called the data code block. The data code block is used by the Get object to read the value of a variable into the Get object's edit buffer and then to write the modified value back into the variable.

When a Get object has input focus, the user can modify the contents of the edit buffer. A Get object uses various methods to control cursor navigation and transfer characters into the edit buffer. Get objects can also perform data validation before and after data input. Rules for data validation are specified in code blocks contained in instance variables of the Get object.

The command @...GET creates a Get object that already contain a code block to access the specified variable. When Get objects are created directly using the class method *Get():new()*, a data code block must be specified. Get objects are stored in an array that is passed to the function ReadModal(). If the @...GET command is used, the array referencing the Get objects is contained in the variable *GetList*. The READ command passes the GetList array to the function ReadModal(). The ReadModal() function is the default edit routine which accesses various edit and display methods of the Get objects. The default Xbase++ Get system includes the ReadModal() function along with the other globally visible utility routines listed in the following table:

Functions of the Get system

Function	Description
READ Service routines	
ReadModal()	Activates data input for all Get objects
ReadExit()	Defines termination keys for data entry fields
ReadInsert()	Toggles between insert and overwrite mode
ReadKill()	Terminates data input for all Get objects in the current GetList array
GET Service routines	
GetEventReader()	Get Reader for AppEvent()
GetHandleEvent()	Allows event processing by current Get object
GetPrevalidate()	Prevalidates before a Get object receives focus
GetPostvalidate()	Postvalidates before a Get object loses focus

Function	Description
GetDoSetkey()	Executes code block which is linked to a key or an event
GetActive()	Sets the Get object that has the focus
GetKillActive()	Removes focus from the active Get object
Getlist()	Determines current GetList array
GetlistPos()	Returns position of current Get object in the GetList array
GetEnableEvents()	Toggles between Inkey() and AppEvent()
GetToMousePos()	Positions cursor in entry field at the mouse pointer

Compatibility functions supporting only keyboard entry

GetReader()	Get reader for Inkey()
GetApplykey()	Allows keys to be processed by the current Get object

Two of the functions in the GETSYS.PRG file exist in order to ensure compatibility with the Clipper Get system. In these functions, the compatibility function Inkey() is used to retrieve keyboard entry. This is not compatible with the AppEvent() function which is used to read Xbase++ events. To make it easier to port existing Clipper applications to Xbase++, the compatibility functions are used by default in VIO mode. This means that the Get system reads only the keyboard using Inkey() and does not process any events. In this default mode, the Get system is completely compatible with Clipper but the mouse is not available.

The default setting is switched when SetMouse(.T.) is called before the first READ command or the first call to ReadModal(). Events in the Get system are then processed using the functions GetEventReader() and GetHandleEvent() instead of the functions GetReader() and GetApplykey(). In this setting the mouse is available. The function SetAppEvent() must be used instead of SetKey() to associate keystrokes with code blocks. The function GetEnableEvents(.T. | .F.) can also be used to switch between the compatibility mode and the event driven mode of the Xbase++ Get system.

The service functions of the Get system are rarely needed. The @...SAY...GET command is sufficient to allow complex data entry screens to be easily programmed. A detailed knowledge of the additional Get system functions is required when the default Get system is modified to meet unique requirements.

14.1.5. Modification of the Get system

The source code of the Xbase++ Get system is contained in the file GETSYS.PRG and can be modified in just about any way. Changes should not be performed in the file itself but in a copy of the file. Before substantial changes to GETSYS.PRG are performed, other ways to modify the default behavior to meet individual requirements should be explored. For example, the instance variable *oGet:reader* contains a code block that calls a user-defined

Get reader that can be used to provide special features. The Get reader code block must accept one parameter (the Get object) which must be passed as the first parameter to the function or procedure that implements the Get reader. The basic approach to implementing a special Get reader is shown in the following example. This code uses the compatibility function Inkey() to read keyboard input. The purpose of the example reader is to handle German language umlauts and "ß" entered as characters in the entry field. These characters are translated into their two character equivalents in the user-defined reader *NoUmlauts()* :

```
#include "Get.ch"

**************
PROCEDURE Main
   LOCAL cName1 := Space(20),  cName2 := Space(20)

   @  8,10 SAY "Input without umlauts and ß"

   @ 10,10 SAY "First Name:" GET cName1 ;    // define Get Reader
                          SEND reader := {|oGet| NoUmlauts(oGet) }
   @ 12,10 SAY " Last Name:" GET cName2 ;
                          SEND reader := {|oGet| NoUmlauts(oGet) }
   READ

RETURN
```

The definition of these input fields occurred using the command syntax. The code block which assigns the user-defined Get reader is specified using the SEND option. In it the reader code block is assigned to the instance variable *oGet:reader*. The Get reader itself is programmed in the following procedure:

```
***************************
PROCEDURE NoUmlauts( oGet )
   LOCAL bBlock, cChar, nKey

   IF GetPreValidate( oGet )          // perform prevalidation
      oGet:setFocus()                 // set input focus to Get

      bBlock := {|o,n1,n2| ;          // translate within the code
         GetApplykey(o,n1),;          // block into two characters
         IIf( o:typeOut, NIL, GetApplykey(o,n2) ) }

      DO WHILE oGet:exitState == GE_NOEXIT

         IF oGet:typeOut              // no editable characters
            oGet:exitState := GE_ENTER // right of the cursor
         ENDIF

         DO WHILE oGet:exitState == GE_NOEXIT
```

```
            nKey := Inkey(0)              // read keyboard
            cChar:= Chr(nKey)

            DO CASE                     // translate special characters
            CASE cChar == "Ä"
               Eval( bBlock, oGet, Asc("A"), Asc("e") )
            CASE cChar == "Ö"
               Eval( bBlock, oGet, Asc("O"), Asc("e") )
            CASE cChar == "Ü"
               Eval( bBlock, oGet, Asc("U"), Asc("e") )
            CASE cChar == "ä"
               Eval( bBlock, oGet, Asc("a"), Asc("e") )
            CASE cChar == "ö"
               Eval( bBlock, oGet, Asc("o"), Asc("e") )
            CASE cChar == "ü"
               Eval( bBlock, oGet, Asc("u"), Asc("e") )
            CASE cChar == "ß"
               Eval( bBlock, oGet, Asc("s"), Asc("s") )
            OTHERWISE
               GetApplykey( oGet, nKey )
            ENDCASE

         ENDDO

         IF ! GetPostValidate( oGet )
            oGet:exitState:= GE_NOEXIT // postvalidation
         ENDIF                         // has failed
      ENDDO

      oGet:killFocus()                 // set back input focus

   ENDIF

RETURN
```

Within the user-defined Get reader *NoUmlauts()*, keys are read using Inkey(). The keyboard input is then tested to see if it includes one of the German language characters Ä, Ö, Ü, ä, ö, ü and ß. If one of these characters is present, it is translated to the two equivalent characters Ae, Oe, Ue, ae, oe, ue, or ss. All other characters are passed along with the Get object to the function GetApplykey() which defines the default behavior for transferring characters into the edit buffer of Get objects.

In order to modify this Get reader to use the function AppEvent() for reading events (instead of just keystrokes), additional variables must be declared for the message parameters passed by reference to AppEvent(). Instead of GetApplykey(), the function GetHandleEvent() must be used for default handling of other events.

14.1.6. Display of tables

Along with formatted data input using @...SAY...GET with READ or using Get objects with ReadModal(), displaying an entire table is an important and frequently used interface feature. A table allows the user to interactively view large amounts of data. Data in a table is organized into rows and columns. This data can be from a database file open in a work area or from an array stored in memory. The TBrowse class and the TBColumn class are included in Xbase++ to allow tables to be displayed. These classes are used together but each performs a specific set of tasks when displaying tables. Table display is possible only through the combined efforts of objects of both classes. A TBrowse object performs the screen display and controls the current row and column of a table. A TBColumn object provides the data for a column of the table displayed by the Tbrowse object. TBColumn objects must be assigned to the TBrowse object and are used by the TBrowse object when a user views a table.

The *TBrowse* and *TBColumn* classes are associated with a set of functions that exist in Xbase++ only to provide compatibility with Clipper. The source code for the compatibility functions listed in the following table are contained in the files DBEDIT.PRG, BROWSYS.PRG and BROWUTIL.PRG.

Functions for the display of tables and database files

Function	Description
Class functions:	
TBrowse()	Return class object of the TBrowse class
TBColumn()	Return class object of the TBcolumn class
Compatibility:	
Browse()	Browse database file, including DELETE and APPEND
DbEdit()	Browse database file with UDF control
TBrowseNew()	Create TBrowse object
TBrowseDb()	Create TBrowse object for database file
TBColumnNew()	Create TBColumn object

TBrowse objects provide a versatile mechanism for displaying data in the form of a table. TBrowse objects display tabular data in a defined section of the screen (the browse window).

Tables are often too large to be entirely displayed on the screen. TBrowse objects provide methods that allow tables to be viewed interactively on the screen. The TBrowse class is designed so that an object of this class does not know anything about the data source it displays. To display data, a TBrowse object relies on one or more objects of the TBColumn class to provide the data for individual columns in the table. A TBrowse object displays the data provided by the TBColumn objects on the screen. Each TBColumn object deals with only one column of the associated table. The TBrowse object manages its own cell cursor and displays the data in the current row and column of a table (the current cell) in a highlighted color.

A user can position the cell cursor in the table using the cursor keys. A TBrowse object includes many methods which move the cell cursor. As soon as the user tries to move the cell cursor out of the Browse window, the TBrowse object automatically synchronizes the visible data on the screen with the data of the underlying table and scrolls the data in the browse window.

Since the data source of the table is not known to the TBrowse object, three functions must be specified to execute the three basic operations that can be performed on a table: jump to the start of the table, jump to the end of the table and change the current row within the table. These operations are comparable to the file commands GO TOP, GO BOTTOM and SKIP. These functions are provided to TBrowse objects as code blocks that are automatically executed within specific methods of the TBrowse object.

The browse window where the TBrowse object displays tabular data can be divided into three areas: headers or column titles, data lines containing the data, and footers at the bottom of each column. Each area, as well as each individual column, can be optionally delimited by a separating line.

In contrast to TBrowse objects, TBColumn objects are very simple objects that contain instance variables but do not have their own methods. TBColumn objects are needed to provide data for tables displayed using TBrowse objects and are useless without an associated TBrowse object. A TBColumn object contains in its instance variable all of the information required for displaying a single column of a table in the browse window of the TBrowse object.

The most important TBColumn instance variable (*:block*) contains a data code block that provides the data from the data source for a column of data. This code block might access a field variable in a work area, or a column in an array. TBColumn objects can also control the color of the data displayed based on its value.

If new values are assigned to the instance variables of a TBColumn object after the TBrowse object has already displayed data using the TBColumn object, the method oTBrowse:configure() must be executed to update the columns in the browse window (see TBrowse class).

14.2. Applications in graphics mode (GUI mode)

This section describes the concepts that are important when programming with Xbase Parts and developing GUI applications. It starts with an overview of Xbase Parts and how events are processed by code blocks or callback methods. The XbpCrt() and XbpDialog() classes that provide objects for managing the application window of an Xbase^{++} program are then detailed. XbpCrt windows are suitable for porting existing DOS programs to a graphic user interface and allow simultaneous text based and graphic output. These windows are used in hybrid applications that transition existing DOS programs to GUI applications in a series of steps. Only graphic output can occur in XbpDialog windows which are thus used only in GUI applications.

14.2.1. Basics of Xbase Parts

Programming graphical user interfaces for applications is easily performed using the Xbase^{++} object model and system resources available on the Xbase^{++} language level. Through its "Xbase Parts" (XBPs), Xbase^{++} offers ways for the programmer still thinking in procedural terms to create programs with graphical user interfaces. Xbase Parts provide graphic dialog elements, like pushbuttons and checkboxes, that can be integrated into character based applications as well as into pure graphic applications. All Xbase Parts are based on operating system resources and fit into the event driven design of the operating system. In order to use Xbase Parts the program must be linked for GUI mode. All Xbase Parts use the same basic mechanisms which are described in this section.

What are Xbase Parts?

XBPs are objects that provide the complex mechanisms of the operating system on the Xbase^{++} language level. A distinction can be made between Xbase Parts (XBPs) designed for screen interaction with the user and those that are used only for graphic output or are not visible. The second group of XBPs is described in the chapter "The Xbase^{++} Graphics Engine (GRA)" and is not discussed in this chapter. The XBPs for user interaction each provide a single dialog element as a graphic component. For this reason, XBPs can only be used in a program that is linked for the GUI mode. In Xbase^{++}, there are Xbase Parts for windows, pushbuttons, checkboxes, data entry fields, etc., as well as standard dialogs for such tasks as selecting fonts or files.

The life cycle of Xbase-Parts

All XBPs are subject to what is called the "life cycle", which distinguishes them from other objects like TBrowse objects or Thread objects. All objects, including XBP objects are created from their class object using the method *:new()*. However, if the object is an XBP object it is not yet capable of doing anything except requesting system resources from the operating system. This is done using the method *:create()*. An XBP is operational only after execution of the *:create()* method. The following program code is an example of this process for a pushbutton:

```
// create object for XBP and specify coordinates
oButton := XbpPushButton():new( , , {10,20}, {80,30} )

// define variables for configuring system resources: in this
// case the text which is displayed on the pushbutton
oButton:caption := " Delete "

// request system resource
oButton:create()
```

These lines show the basic process for creating an operational XBP. The object is first created using the class method *:new()*. Then values are assigned to the instance variables used to configure the system resource, and finally *:create()* is used to request the system resource from the operating system (in this case, a pushbutton). It is important to distinguish between *:new()* and *:create()*. Both methods must be called to generate an operational Xbase Part. The *:new()* method generates the object and the *:create()* method requests system resources. Most XBPs use instance variables only to configure the system resource. Assigning values to instance variables used for configuration must occur before the *:create()* method is called.

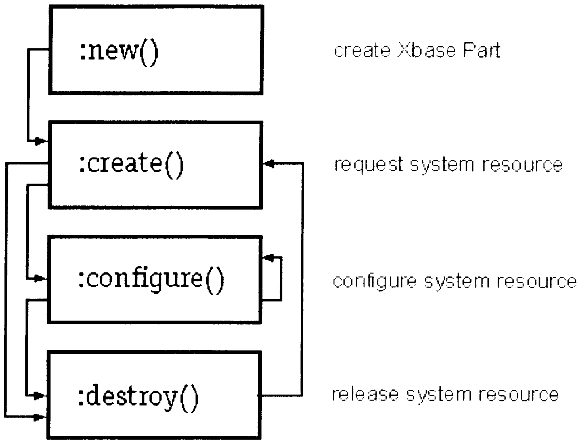
The system resource can be reconfigured after an XBP is created and has requested the system resource from the operating system. If new values are assigned to the instance variables used for configuration of the system resource, the method *:configure()* must be called to implement the changes. The following shows an example:

```
// reconfigure a loaded system resource
   oButton:caption := " Recall "
   oButton:configure()
```

The text (or caption) displayed on the pushbutton is part of the "pushbutton" system resource. In order to change it, the system resource must be reconfigured. The XBP method *:configure()* is used to accomplish this (and is one of the "life cycle" methods of an XBP).

The last method in the "life cycle" of an XBP is *:destroy()*. This method releases the system resources that were requested from the operating system by the method *:create()*. This renders the XBP non-operational but it remains in existence and can request system resources again. In other words, the method *:destroy()* has no influence on the object created using *:new()*, but releases the system resources requested using *:create()*.

The following diagram illustrates the "life cycle" of an Xbase Part and clarifies when each of the methods are called:



The method *:destroy()* is generally not needed. However, it can be used to explicitly release memory intensive system resources (for example, bitmaps). System resources are implicitly released as soon as there are no further references to the Xbase Part object. In this case, an Xbase Part is handled like any other memory variable and is removed from memory by the garbage collector.

Important: As long as the *:create()* method has not been called an XBP is unable to execute any other method (there are some rare exceptions which are explicitly outlined). Immediately after the call to *:new()* it is only possible to assign values to instance variables which are used to configure system resources. Only after the system resource has been retrieved from the operating system using *:create()* an XBP is fully functional. Then it can execute all available methods except *:create()*. This method can be called again only after a call to the *:destroy()* method which releases system resources.

Xbase Parts and events

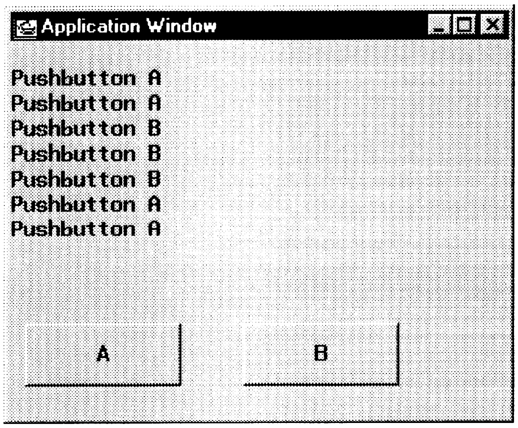
XBPs are seamlessly inserted into the message stream used to control a GUI application. Messages identify the events that have taken place. Examples of events that a program would receive are keyboard entry and mouse clicks. Events such as these are read using the function AppEvent(), which returns a numeric event code (see "Keyboard and mouse" in the section "Applications in character mode" of this chapter). The function AppEvent() plays a central role in controlling the Xbase Parts that provide graphic dialog elements.

The following program code shows the basic relationship:

```
#include "Appevent.ch"

PROCEDURE Main
   LOCAL nEvent, mp1, mp2, oXbp

   // create first pushbutton
   oXbp:= XbpPushButton():new()
   oXbp:caption := "A"
   oXbp:create( , , {10,20}, {100,40} )
   oXbp:activate := {|| QOut( "Pushbutton A" ) }

   // create second pushbutton
   oXbp := XbpPushButton():new()
   oXbp:caption := "B"
   oXbp:create( , , {150,20}, {100,40} )
   oXbp:activate := {|| QOut( "Pushbutton B" ) }

   // Event loop
   nEvent := 0
   DO WHILE nEvent <> xbeP_Close
      nEvent := AppEvent( @mp1, @mp2, @oXbp )
      oXbp:HandleEvent( nEvent, mp1, mp2 )
   ENDDO
RETURN
```

In this example, two pushbuttons are created and the entire program control lies within a single DO WHILE loop (the event loop). In this event loop, events are read from the event queue using AppEvent(). The third parameter passed by reference to the function is assigned a reference to the addressee for the event. The addressee might be the window or one of the pushbuttons. The event is processed in the *:handleEvent()* method of this addressee object. Each pushbutton has an "activate" code block that is automatically executed within the *:handleEvent()* method when a pushbutton is clicked with the mouse. Within these code blocks a character is output on the screen using QOut().

The following illustration shows what the output on the screen might look like if each of the two pushbuttons were clicked several times:



```
Pushbutton A
Pushbutton A
Pushbutton B
Pushbutton B
Pushbutton B
Pushbutton A
Pushbutton A
```

Output from the example program

An important point shown in the example program is that the function AppEvent() in the event loop not only reads the event from the event queue, but also determines the addressee for the event. The addressee is always an Xbase Part and each XBP includes the method *:handleEvent()*. Three parameters are always passed by reference to the *:handleEvent()* method: the numeric event code and the two message parameters. When the message is processed within the *:handleEvent()* method, the code block stored in the *:activate* instance variable of the pushbutton is executed. This code block contains the XBP's reaction to the specific event. When the left mouse button is clicked on a pushbutton, the xbeP_Activate event is generated and the numeric event code (corresponding to a #define constant) is returned by AppEvent(). When this numeric code is passed on to *:handleEvent()*, the method executes the code block contained in the instance variable *:activate.*

For each XBP there are many instance variables that can contain code blocks that are executed in response to specific events. The event is first read from the event queue using AppEvent(). Then the reaction occurs when the code block provided for the event is executed. This sequence of events is called a "callback" approach. For this reason, all instance variables that contain code blocks defining reactions to events are referred to as "callback slots". A callback slot contains either the value NIL or a code block to be executed by the *:handleEvent()* method when the specific event occurs. The following table lists the#define constants of predefined events for user interaction and the corresponding callback slots available for all XBPs:

Default events which are processed by Xbase Parts

Event code	Callback slot	Description
Mouse events		
xbeM_LbClick	:LbClick	Click left button
xbeM_LbDblClick	:LbDblClick	Double click left button
xbeM_LbDown	:LbDown	Left button pressed
xbeM_LbUp	:LbUp	Left button released
xbeM_MbClick	:MbClick	Click middle button
xbeM_MbDblClick	:MbDblClick	Double click middle button
xbeM_MbDown	:MbDown	Middle button pressed
xbeM_MbUp	:MbUp	Middle button released
xbeM_Motion	:Motion	Mouse moved
xbeM_RbClick	:RbClick	Click right button
xbeM_RbDblClick	:RbDblClick	Double click right button
xbeM_RbDown	:RbDown	Right button pressed
xbeM_RbUp	:RbUp	Right button released
Other events		
xbeP_KeyBoard	:KeyBoard	Keyboard entry occurred
xbeP_HelpRequest	:HelpRequest	Help requested
xbeP_SetInputFocus	:SetInputFocus	Input focus granted
xbeP_KillInputFocus	:KillInputFocus	Input focus lost
xbeP_Move	:Move	XBP moved
xbeP_Paint	:Paint	XBP redrawn
xbeP_Quit	:Quit	Application terminated
xbeP_Resize	:Resize	Size of XBP changed

The callback slots in this table are included in all XBPs and allow user interaction by processing events. Many XBPs have additional callback slots for events that only occur with specific XBPs. For example, pushbuttons have the *:activate* callback slot, that can contain a code block to execute in response to the xbeP_Activate event. This callback slot is not present in other XBPs.

In summary, the example program above shows the basic process for programming using Xbase Parts. An XBP is created and receives the "knowledge" of how it should react to events through the code blocks that are assigned to the callback slots. The *:handleEvent()* method processes events and executes the corresponding code block. The program is controlled in the event loop where events are read using AppEvent(). This function also determines which Xbase Part is to process each event. In the end, the application is controlled entirely by the user who generates the events.

Callback methods for processing events

The preceding section presents the basic approach to processing events with XBPs and describes the simple model where the reaction to an event is defined by a code block assigned to one of the predefined callback slots. Xbase++ also offers a second model. Although this model is more complicated, it has broader capabilities for processing events. This approach utilizes callback methods. Along with each callback slot, Xbase Parts contains a callback method of the same name that is executed instead of the code block contained in the callback slot. Like the callback code block, the corresponding callback method is also executed by the method *:handleEvent()* for each event it receives. Callback methods within the predefined XBPs do not execute any code. They are available for user-defined XBPs where the reaction to an event is programmed as a method rather than as a code block. The following example program demonstrates this approach. Its effect is the same as the example in the previous section and it displays the same results on the screen:

```
#include "Appevent.ch"

PROCEDURE Main
   LOCAL nEvent, mp1, mp2, oXbp
   SetColor("N/W")
   CLS

   // create user-defined pushbuttons
   MyPushButton():new( "A", { 10,20}, {100,40} ):create()
   MyPushButton():new( "B", {150,20}, {100,40} ):create()

   // Event loop
   nEvent := 0
   DO WHILE nEvent <> xbeP_Close
      nEvent := AppEvent( @mp1, @mp2, @oXbp )
      oXbp:HandleEvent( nEvent, mp1, mp2 )
   ENDDO
RETURN

// user-defined pushbutton
CLASS MyPushbutton FROM XbpPushbutton
   EXPORTED:
   METHOD init, activate
ENDCLASS

// initialize superclass and self
METHOD MyPushbutton:init( cCaption, aPos, aSize )
   ::XbpPushButton:init(,, aPos, aSize )
   ::caption := cCaption
RETURN self
```

```
// callback method for the event xbeP_Activate
METHOD MyPushbutton:activate
   QOut( "Pushbutton ", ::caption )
RETURN self
```

This program differs from the example in the previous section in a couple of ways. The Main procedure is abbreviated and the MyPushButton class is instantiated instead of the XbpPushButton class. The method *:activate()* has been redefined in the user-defined class MyPushButton to produce output using QOut(). In the previous example, the output was programmed within callback code blocks. The role of the callback code blocks in the previous example is replaced by the callback method *:activate()* in this example. The method *:activate()* is then executed when the left mouse button is clicked on either one of the two pushbuttons. Callback methods are used to process events in user-defined Xbase Parts that are inherited from existing XBPs. They generally replace the code in callback code blocks and require that the new class be derived from an existing Xbase Part. The callback method of the new class contains the program code that is executed when a specific event occurs. When a callback method is defined, a code block generally should not also be assigned to the callback slot of the same name, since it would then be executed after the callback method.

Instance variables and methods existing in all Xbase Parts

All Xbase Parts, even those that do not process events or have a visual representation, include the instance variable *:cargo*. This instance variable is not used by any Xbase Part but allows user-defined data to be contained in the XBP without requiring that a new class be created.

XBPs also include the *:status()* method, which returns the current condition of an XBP in its life cycle. This method returns a numeric value corresponding to a #define constant from the XBP.CH file. The possible return values are listed in the following table:

Constants for the return value of :status()

Constant	Description
XBP_STAT_INIT	Xbase Part initialized
XBP_STAT_CREATE	System resources request successful
XBP_STAT_FAILURE	System resources could not be provided

The method *:status()* is mainly useful for debugging purposes. It can be called after the method *:create()* to determine whether the system resources were provided for an Xbase Part.

14.2.2. Windows and relationships

All output in a GUI application occurs in a window on the screen. The general meaning of the term "window" is a rectangular screen area containing a graphic display. The application window visible on the screen actually consists of a frame and a number of additional windows. All Xbase Parts that have a visible representation are windows in the sense of this broader definition.

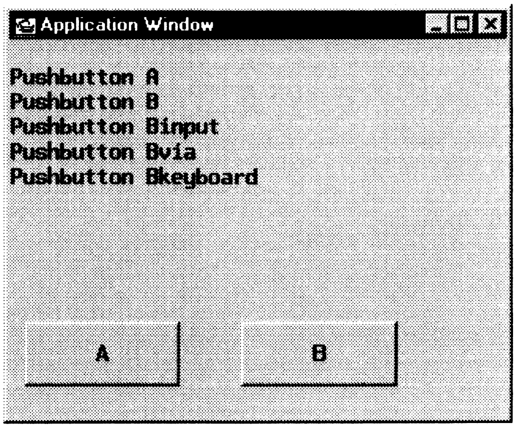
The program itself runs in an application window which limits the area that an application uses on the screen. The application window contains dialog elements (Xbase Parts) that also represent windows. Each GUI application is made up of many windows. A hierarchical relationship exists between these windows and is described using the terms "parent" and "child". A parent window provides the display area for child windows and child windows are contained in the parent window.

The parent-child relationship between windows represents a physical relationship and is the most important relationship in programming using Xbase Parts. Returning to the example from the previous section, the pushbuttons are children of the application window and are contained in it. Viewed from the other direction, the application window is the parent of the two pushbuttons and contains the Xbase Parts. This relationship is very significant in processing keyboard events. Whenever a child XBP receives a key that it is not able to process or does not understand, the event is sent on to the parent for processing. A modification of the example in the previous section illustrates this relationship:

```
#include "Appevent.ch"

PROCEDURE Main
   LOCAL nEvent, mp1, mp2, oXbp
   SetColor("N/W")
   CLS

   // get application window
   oXbp := SetAppWindow()

   // display character corresponding to key pressed
   oXbp:KeyBoard := {|mp1| QQOut( Chr(mp1) ) }

   // create user-defined pushbuttons
   MyPushButton():new( "A", { 10,20}, {100,40} ):create()
   MyPushButton():new( "B", {150,20}, {100,40} ):create()

   // Event loop
   nEvent := 0
   DO WHILE nEvent <> xbeP_Close
      nEvent := AppEvent( @mp1, @mp2, @oXbp )
      oXbp:HandleEvent( nEvent, mp1, mp2 )
   ENDDO
RETURN
```

In this example, the application window is determined using the function SetAppWindow(). The application window is assigned a callback code block for the event xbeP_Keyboard. Any keys that the pushbuttons can not process are output as characters in the window.

The display on the screen might appear as follows:



```
Pushbutton A
Pushbutton B
Pushbutton Binput
Pushbutton Bvia
Pushbutton Bkeyboard
```

Display of example program

Pushbutton "B" has input focus. A pushbutton processes the space bar itself and is activated by this key. In the example the string "Input via keyboard" was entered and only the blank spaces were recognized by the pushbutton. Any other characters input from the keyboard are not understood by the pushbutton, and are instead processed by the parent of the pushbutton. The callback code block assigned to the *:keyBoard* instance variable in the application window uses QQOut() to display the keyboard character that was pressed.

The child list

Relations between windows, or Xbase Parts, respectively, are managed by the XbpPartHandler class. It is the root class for all XBPs having a visual representation. When an XBP executes the *:create()* method, the object is automatically added to an internal array of its parent window: the child list. The child list references all XBPs contained in a window (Note: the child list array is returned by the *:childList()* method). Therefore, in a program it is sufficient to assign the reference of the application window to a variable in order to be able to access all contained XBPs. These are added to the child list array in the same order as they execute the *:create()* method. If the creation order is known an XBP can be retrieved using a numeric index. For example:

```
oButton := oXbp:childList()[2]
? oButton:caption                    // result: B
```

In the example program, the pushbutton "B" is created second. It is referenced in the second element of the child list array of the application window.

Overlapping XBPs

On the one hand, the creation order of XBPs determines their position in a parent window's child list. On the other hand, it determines the order of display. The XBP which executes its *:create()* method first is displayed first, too. The one created last is displayed last. Therefore, it appears in front of all other XBPs. This aspect becomes important when XBPs have the same positions in a window or are overlapping each other. This is the case when XBPs are grouped visually by a surrounding frame. For instance, if single line entry fields (XbpSLE) are to be grouped by a static frame (XbpStatic), the frame must be created first and then the entry fields are to be positioned inside the frame.

Additional information about the relationships between windows can be found in the reference documentation for this class. At this point, two Xbase Parts which may be used as the first or highest parent in the parent-child hierarchy of a GUI application need to be described first. These provide the application window. The application window manages the area on the screen where all output of a GUI application occurs and where all other Xbase Parts are displayed (an exception is MDI applications which can have several application windows). The application window for a GUI application is created in the function AppSys() when the Xbase^{++} program starts. The source code for AppSys() is contained in the file ..\SOURCE\SYS\APPSYS.PRG. A window of the XbpCrt class is generated by default and allows both text-based and graphic output. However, the function AppSys() can be modified so that the application window is provided as an object of the XbpDialog class. In this case, only graphic output is possible.

14.2.3. XbpCrt() - The window for hybrid mode

The default application window used by a GUI application is an XbpCrt window. This type of window is a hybrid between text mode (VIO mode) and graphics mode. It was created for Xbase^{++} to allow a seamless migration from DOS applications (Clipper programs) to a 32bit operating system with graphic user interface. Within an XbpCrt window, data can be output using pure text oriented functions like DispOut() or QOut() as well as pure graphic output using functions of the GRA engine. An XbpCrt window combines the text mode capabilities of a VIO application with the graphics mode capabilities of a GUI application. This combined form is called "hybrid mode", and allows stepwise porting of existing DOS text mode applications into 32bit GUI applications.

An XbpCrt window allows graphic and text oriented output. Also, Xbase Parts can be displayed in XbpCrt windows so that a program originally developed under DOS can be transitioned step by step into a GUI application. Special adaptations for processing events are not required, since an XbpCrt window performs these tasks independently. As an example, consider a simple program created using a purely procedural approach that implements data entry using a series of Get objects.

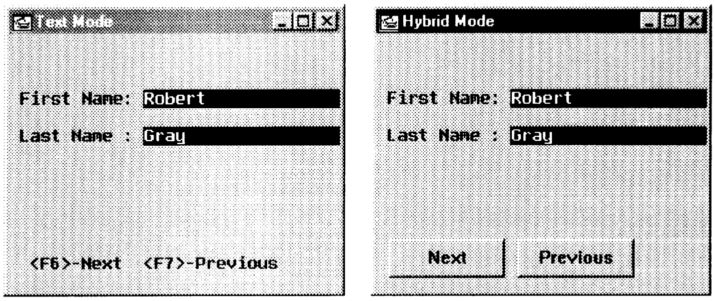
The current record is changed during input via the F6 and F7 function keys:

```
#include "Inkey.ch"

PROCEDURE Main

   SetColor( "N/W,W+/N" )
   CLS
   @ MaxRow()-1, 2 SAY "<F6>-Next  <F7>-Previous"

   SetKey( K_F6, {|| DbSkip( 1), RefreshGetList() } )
   SetKey( K_F7, {|| DbSkip(-1), RefreshGetList() } )

   USE Customer EXCLUSIVE

   DO WHILE LastKey() <> K_ESC
      @ 3,1 SAY "First Name:" GET Customer->FIRSTNAME
      @ 5,1 SAY "Last Name :" GET Customer->LASTNAME
      READ
   ENDDO
RETURN

PROCEDURE RefreshGetList
   AEval( GetList, {|oGet| oGet:reset() , ;
                            oGet:display() } )
RETURN
```

The program logic for changing records is implemented via code blocks that are associated with function keys using SetKey(). The function keys that are linked to code blocks are ideal candidates for replacement with pushbuttons in a hybrid mode application. This would help give the text mode application a more GUI look. Pushbuttons are easily added to the XbpCrt window. The two pushbuttons just need to be created in the Main procedure and replace the function SetKey():

```
#include "Inkey.ch"

PROCEDURE Main
   LOCAL oXbp
   SetColor( "N/W,W+/N" )
   CLS

   SetMouse(.T.)
   oXbp:= XbpPushButton():new()
   oXbp:caption := "Next"
   oXbp:activate:= {|| DbSkip( 1), RefreshGetList() }
   oXbp:create( ,, {10,10}, {90,30})

   oXbp:= XbpPushButton():new()
   oXbp:caption  := "Previous"
   oXbp:activate:= {|| DbSkip(-1), RefreshGetList() }
   oXbp:create( ,, {110,10}, {90,30})

   USE Customer EXCLUSIVE

   DO WHILE LastKey() <> K_ESC
      @ 3,1 SAY "First Name:" GET Customer->FIRSTNAME
      @ 5,1 SAY "Last Name :" GET Customer->LASTNAME
      READ
   ENDDO
RETURN
```

The result of this change in the example program is shown in the next illustration:



Migration from text mode to a hybrid application

There are a couple of important points in these two examples. First, only control of database movement is handled by objects of the XbpPushButton() class. The actual logic of the data input screen is not changed and the program still appears to be procedural. It is now somewhat more event driven, but only the task of handling pushbutton events is assumed by the XbpCrt window. The call to the function SetMouse(.T.) at the start of the second example causes the Get entry fields to react to mouse events. This means that within the Get system events are retrieved using AppEvent() instead of the compatibility function Inkey(). This process offers a very easy way to port existing DOS applications into a GUI environment in a stepwise manner. As another example, all @...PROMPT / MENU TO commands for program control are easily transitioned to a menu system with a main menu appearing as a menu bar at the top of the XbpCrt window.

The modified program shows another characteristic of an XbpCrt window: text mode row and column coordinates with an origin in the upper left corner of the window can be used as well as graphic xy coordinates with the origin at the bottom left (the pushbuttons are positioned in the window based on xy coordinates). Within an XbpCrt window dialog functions like AChoice() or MemoEdit() can be used and their output can be displayed next to graphic output such as Xbase Parts, bitmaps or a bar chart. The XbpCrt window also supports the text mode color system along with graphics mode colors which means that colors can be set using SetColor() or GraSetColor().

Since an XbpCrt window also allows text oriented output, its maximum size is limited to the function values of MaxRow() and MaxCol(). The default window size is 25 text lines and 80 columns corresponding to the size of a text screen. The size of the screen can be reduced using the mouse, but it can not be made larger than MaxRow() and MaxCol(). Changing the maximum number of rows and columns can be done using the Clipper compatible function SetMode(). SetMode() supports a limited number of sizes. Alternatively, the size of a text mode window can be set when creating an XbpCrt window to any number of rows and columns desired.

In summary, all commands and functions in the scope of the Clipper language and the additional functionality of Xbase++ and Xbase Parts can be used in an XbpCrt window. An XbpCrt window is created for the exclusive purpose of allowing an easy way to port existing DOS applications to the GUI environment. This has the advantage of allowing the seamless migration of existing text mode applications to GUI. The disadvantage of this approach is the increased memory requirement of an XbpCrt window as opposed to a dialog window that does not allow text based output. The increased memory requirement of an XbpCrt window is inevitably paid for in a loss of performance. For this reason, it is recommended that new development of applications under Xbase++ not use the XbpCrt window. This is accomplished by modifying the function AppSys() in APPSYS.PRG to create an object of the XbpDialog() class as the application window.

14.2.4. XbpDialog() - The window for GUI mode

The class XbpDialog() provides an application window for pure graphic mode. Text based output is not possible in these dialog windows. This means that all output commands and functions that perform text based screen output under Clipper can no longer be used. Screen output occurs in a dialog window using only Xbase Parts and the functions of the GRA engine. XbpDialog objects are used to program pure GUI applications that are optimized for the operating system and use most of its capabilities.

Before continuing the discussion of the XbpDialog object, it is worthwhile to clarify which functions and commands are no longer available. The commands and functions listed in the following table can not be used if the current window is an XbpDialog object:

Commands and functions that cannot be used in dialog windows

Commands		Functions	
? | ??	SET CURSOR	AChoice()	QOut() | QQOut()
@...BOX	SET DELIMITERS	Alert()	Read...()
@...CLEAR	SET INTENSITY	Browse()	RestScreen()
@...GET	SET MESSAGE	Col()	Row()
@...PROMPT	SET SCOREBOARD	ColorSelect()	SaveScreen()

Commands		Functions	
@...SAY	SET WRAP	DbEdit()	Scroll()
@...TO	TEXT	DevOut()	SetBlink()
ACCEPT	TYPE	DevOutPict()	SetColor()
CLEAR ALL	WAIT	DevPos()	SetCursor()
CLEAR GETS		DispBegin()	SetMode()
CLEAR SCREEN		DispCount()	SetMouse()
DISPLAY		DispEnd()	SetPos()
INPUT		DispOut()	TBApplyKey()
LIST		DispOutAt()	TBColumn()
MENU TO		Get()	TBColumnNew()
READ		Get...()	TBHandleEvent()
RESTORE SCREEN		GetNew()	TBrowse()
SAVE SCREEN		MaxCol()	TBrowseDb()
SET COLOR		MaxRow()	TBrowseNew()
SET CONSOLE		MemoEdit()	TBtoMousePos()

All of the commands and functions affecting the cursor are no longer available, since the cursor as it is used in text mode does not exist in graphics mode. Similarly, the commands and functions that manipulate the text mode screen (such as SaveScreen(), DispBegin(), Scroll() and RESTORE SCREEN) can not be used in a dialog window which operates in graphics mode. Xbase Parts and functions of the GRA engine are used in place of these text based output functions and commands.

Along with these commands and functions there are two other small groups of functions. The first represents compatibility functions that should no longer be used. The second group of functions only operate if the printer has been set as the output device using the command SET DEVICE TO PRINTER. The following table gives an overview:

Commands and functions which are not to be used in dialog windows

Command/Function	Only usable with SET DEVICE TO PRINTER
SET KEY	DevOut()
Inkey()	DevOutPict()
SetKey()	DevPos()
LastKey()	PRow()
NextKey()	PCol()
IsPrinter()	

In order for the application window to be provided as a graphics dialog window, an XbpDialog object needs to be created in the function AppSys(). This function is normally executed prior to the call of the MAIN procedure. The default source code for AppSys() appears in the APPSYS.PRG file, but it can be included in another file if it is linked to the executable file. The following lines are taken from the example program SDIDEMO.PRG (the example is shortened) and give an example of modifications made to AppSys():

```
********************************************************************
* Example for AppSys() in an SDI application
* The function is executed prior to Main()
********************************************************************
PROCEDURE AppSys
   LOCAL oDlg, oXbp, aPos[2], aSize, nHeight:=400, nWidth := 615

   // Get size of desktop window
   // to center the application window
   aSize    := SetAppWindow():currentSize()
   aPos[1]  := Int( (aSize[1]-nWidth ) / 2 )
   aPos[2]  := Int( (aSize[2]-nHeight) / 2 )

   // Create application window
   oDlg := XbpDialog():new()
   oDlg:title := "Toys & Fun Inc. [Xbase++ - SDI Demo]"
   oDlg:border:= XBPDLG_THINBORDER
   oDlg:create( ,, aPos, {nWidth, nHeight},, .F. )

   // Set background color for drawing area
   oDlg:drawingArea:SetColorBG( GRA_CLR_PALEGRAY )

   // Select font
   oDlg:drawingArea:SetFontCompoundName( "8.Helv.normal" )

   // Display application window and set focus to it
   oDlg:show()
   SetAppWindow( oDlg )
   SetAppFocus ( oDlg )

RETURN
```

In this procedure, a reference to the desktop window is returned by the function SetAppWindow() and is used to determine its size. This allows the Xbase++ application window to be centered on the screen when opened, regardless of the screen resolution. The dialog window created in the example has a fixed size, because the box type XBPDLG_THINBORDER does not allow the user to change the size of the window. This is the simplest variation, since changing the size of the application window also involves repositioning or redimensioning the dialog elements (XBPs) contained in the window.

After the dialog window has been created, it is recommended that the background color for the drawing area be set. This also sets the background color for text displayed as captions for XBPs such as XbpStatic, XbpRadioButton or XbpCheckBox. The method *:setColorBG()* returns the color of the drawing area and sets the default color used for displaying all XBPs with captions.

The drawing area of XbpDialog

The background color is set using the method *:setColorBG()*. This method is not executed by the XbpDialog object in the example, but by an object contained in the instance variable *:drawingArea*. This is because the XbpDialog objects contain additional objects of the XbpIWindow class. Objects of this class provide implicit windows that do not have a title bar or a frame, but that manage rectangular screen areas. An XbpDialog object basically consists of a frame that limits the screen area which is available for an application. The title bar of the dialog window as it appears on the screen is a window, as is the drawing area within the dialog. Both these windows are implicit windows of the XbpDialog object. They are objects of the XbpIWindow class and are contained in the instance variables *:titleBarArea* and *:drawingArea*. The drawing area of the XbpDialog object plays an important role when programming with dialog windows.

Important: Each dialog element (XBP) that is displayed in a dialog window must have *:drawingArea* as the parent rather than the XbpDialog object itself. Otherwise the sizeable border of the dialog window becomes the parent and an XBP would be visible only in this 2-8 pixel wide frame.

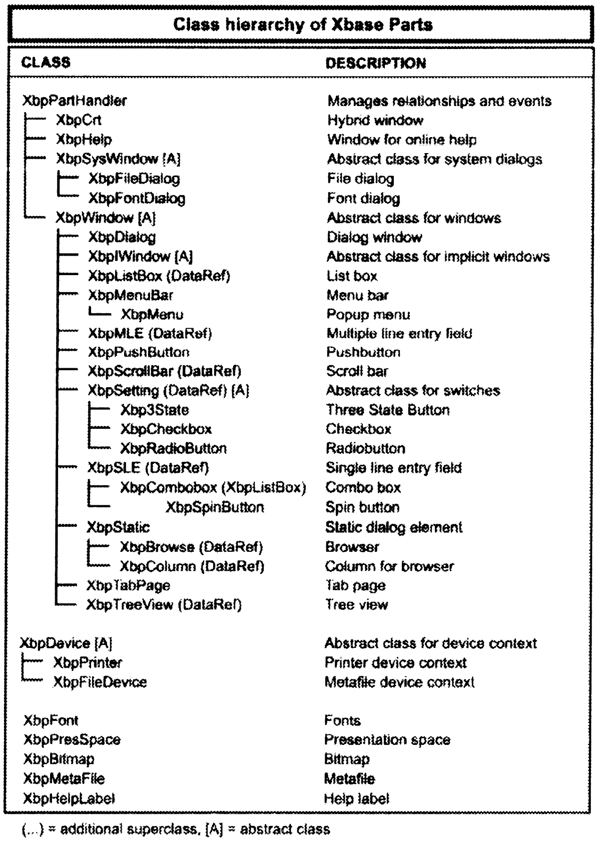
For example, if a pushbutton is to be displayed in a dialog window, it must be a child of the drawing area (*:drawingArea*) of the XbpDialog object. This is a difference between how a dialog window works and an XbpCrt window which can itself act as parent. An XbpDialog object is the parent of only the two implicit windows that manage the drawing area and the title bar. Both of these windows are accessible through instance variables of the XbpDialog object. An example of displaying a pushbutton in a dialog window is shown in the following program code:

```
#include "Appevent.ch"

PROCEDURE Main
   LOCAL nEvent, mp1, mp2, oXbp

   // Create pushbutton
   oXbp:= XbpPushButton():new( SetAppWindow():drawingArea )
   oXbp:caption := "Cancel"
   oXbp:create( , , {10,20}, {100,40} )
   oXbp:activate := {|| PostAppEvent( xbeP_Close) }

   // Event loop
   nEvent := 0
   DO WHILE nEvent <> xbeP_Close
      nEvent := AppEvent( @mp1, @mp2, @oXbp )
      oXbp:HandleEvent( nEvent, mp1, mp2 )
   ENDDO
RETURN
```

Here the parent for the pushbutton is specified in the call to a method. The function SetAppWindow() returns the XbpDialog object created in AppSys() and a reference to its drawing area is passed to the pushbutton method *:new()*. If the drawing area is not specified as the parent for Xbase Parts, it inevitably leads to an error in the display or no display at all within the dialog window. It is important to remember that with XbpDialog objects the drawing area (*oXbpDialog:drawingArea*) must be set as the parent, not the XbpDialog object itself.

14.2.5.Class hierarchy of Xbase Parts

This section gives an overview of the class hierarchy of Xbase Parts. The relationship between different Xbase Parts is illustrated in the illustration below as well as which XBPs utilize the capabilities of another class. The classes marked with "[A]" are abstract classes which can not themselves be instantiated on the Xbase^{++} language level. All Xbase Parts that have a visible representation are derived from the XbpPartHandler class. The XbpPartHandler class implements the parent-child relationship and defines the method *:handleEvent()*.



Class hierarchy of Xbase Parts

CLASS	DESCRIPTION
XbpPartHandler	Manages relationships and events
├── XbpCrt	Hybrid window
├── XbpHelp	Window for online help
├── XbpSysWindow [A]	Abstract class for system dialogs
│ ├── XbpFileDialog	File dialog
│ └── XbpFontDialog	Font dialog
└── XbpWindow [A]	Abstract class for windows
├── XbpDialog	Dialog window
├── XbpIWindow [A]	Abstract class for implicit windows
├── XbpListBox (DataRef)	List box
├── XbpMenuBar	Menu bar
│ └── XbpMenu	Popup menu
├── XbpMLE (DataRef)	Multiple line entry field
├── XbpPushButton	Pushbutton
├── XbpScrollBar (DataRef)	Scroll bar
├── XbpSetting (DataRef) [A]	Abstract class for switches
│ ├── Xbp3State	Three State Button
│ ├── XbpCheckbox	Checkbox
│ └── XbpRadioButton	Radiobutton
├── XbpSLE (DataRef)	Single line entry field
│ ├── XbpCombobox (XbpListBox)	Combo box
│ └── XbpSpinButton	Spin button
├── XbpStatic	Static dialog element
│ ├── XbpBrowse (DataRef)	Browser
│ └── XbpColumn (DataRef)	Column for browser
├── XbpTabPage	Tab page
└── XbpTreeView (DataRef)	Tree view
XbpDevice [A]	Abstract class for device context
├── XbpPrinter	Printer device context
└── XbpFileDevice	Metafile device context
XbpFont	Fonts
XbpPresSpace	Presentation space
XbpBitmap	Bitmap
XbpMetaFile	Metafile
XbpHelpLabel	Help label

(...) = additional superclass, [A] = abstract class

14.2.6. DataRef() - The connection between XBP and DBE

In the class hierarchy of Xbase Parts, nine classes are derived not only from XbpPartHandler or XbpWindow but also from the DataRef class. The DataRef class provides objects that reference data. It provides data from either a memory variable or a field variable for use in an XBP. All Xbase Parts that can modify data also have DataRef as a superclass. The DataRef class provides services for accessing data. In the simplest case, this data is a value that only exists in the edit buffer of an Xbase Part and is changed there. The following Xbase Parts use an edit buffer for storing data:

Xbase-Parts derived from DataRef (XBPs with edit buffer)

XBP	Data type in the edit buffer
XbpSLE	Unformatted character string (single line)
XbpMLE	Formatted character string (multiline)
Xbp3State	Numeric value (0, 1, 2)
XbpCheckBox	Logical value
XbpRadioButton	Logical value
XbpSpinButton	Numeric value (in the limits Min and Max)
XbpListBox	Array (numeric position of the selected entries)
XbpComboBox	Array and character string (special case)
XbpScrollBar	Numeric value (-2^15 to +2^15)

Access to data stored in the edit buffer of an Xbase Part occurs in the method *:getData()* which is implemented in the DataRef class. *:getData()* returns the value in the edit buffer. In the following example, an entry field is created and after each keypress the current contents of the edit buffer are displayed:

```
#include "Appevent.ch"

PROCEDURE Main
   LOCAL nEvent, mp1, mp2, oXbp

   CLS                                   // This is an XbpCrt window

   // Create entry field
   oXbp:= XbpSLE():new()
   oXbp:create( , , {50,50}, {100,30} )
   oXbp:keyBoard := {|mp1, mp2, obj|;
       IIf( mp1== xbeK_ESC, ;            // Esc key was pressed
            PostAppEvent( xbeP_Close), ; // terminate program
            QOut( obj:getData() )     ) } // display buffer

   // Event loop
   nEvent := 0
   DO WHILE nEvent <> xbeP_Close
```

```
      nEvent := AppEvent( @mp1, @mp2, @oXbp )
      oXbp:HandleEvent( nEvent, mp1, mp2 )
   ENDDO
RETURN
```

In the example, the contents of the edit buffer of the XbpSLE object are read in the code block assigned to the *:keyBoard* callback slot. This code block is evaluated after each keypress and receives the XbpSLE object as the third parameter *obj*.

The example illustrates that managing and changing data in a program just requires Xbase Parts that include an edit buffer. The value to be edited is stored in this edit buffer. Generally, no other memory variable referencing the value is needed.

The DataRef class provides the instance variable *:dataLink* to link an Xbase Part to a database field in order to display or change field variables. A data code block that defines how to access a database field can be assigned to this instance variable. This data code block provides the connection between an Xbase Part, a field variable and DatabaseEngine. The following example uses an XbpSLE object to access a database field:

```
#include "Appevent.ch"

PROCEDURE Main
   LOCAL nEvent, mp1, mp2, oXbp, oSLE
   FIELD LASTNAME

   // open database (for exclusive access)
   USE Customer EXCLUSIVE

   // create entry field
   oSLE := XbpSLE():new()
   oSLE:create( ,, {50,100}, {100,30} )

   // reference database field
   oSLE:dataLink := {|x| IIf( x==NIL, LASTNAME, LASTNAME:=x ) }

   // copy database field into SLE
   oSLE:setData()

   // Pushbutton to cancel program
   oXbp := XbpPushButton():new()
   oXbp:caption := "Cancel"
   oXbp:create( , , {50,50}, {80,30} )
   oXbp:activate := {|| PostAppEvent( xbeP_Close ) }

   // Pushbutton to go to next record
   oXbp := XbpPushButton():new()
   oXbp:caption := "Next"
   oXbp:create( ,, {150,50}, {80,30} )
   oXbp:activate := {|| oSLE:getData(), ;  // write data
                        DbSkip( 1 )    , ;  // skip forwards
                        oSLE:setData()  }  // read new data

   // Pushbutton: go to previous record
   oXbp := XbpPushButton():new()
   oXbp:caption := "Previous"
   oXbp:create( ,, {250,50}, {80,30} )
   oXbp:activate := {|| oSLE:getData(), ;  // write data
                        DbSkip( -1 )   , ;  // skip backwards
                        oSLE:setData()  }  // read new data

   // Event loop
   nEvent := 0
   DO WHILE nEvent <> xbeP_Close
      nEvent := AppEvent( @mp1, @mp2, @oXbp )
      oXbp:HandleEvent( nEvent, mp1, mp2 )
   ENDDO
RETURN
```

The example is a little more complex because it creates three kinds of dialog elements: an XbpSLE object that can edit a database field, a pushbutton to cancel the program and two pushbuttons to perform database navigation. One of the most important elements in this example is the data code block assigned to the instance variable *:dataLink*. This code block defines access to the database field LASTNAME. This code block is executed within the methods *:setData()* and *:getData()* which are implemented in the DataRef class (the XbpSLE class also inherits from DataRef!). The method *:setData()* evaluates the data code block without passing a parameter to it and assigns the return value of the code block to the edit buffer of the XbpSLE object. In other words, the method *:setData()* reads the value of the database field LASTNAME and copies it into the edit buffer of the entry field. The method *:getData()* performs the reverse function: it reads the value from the edit buffer of the XbpSLE object and passes this value to the data code block in the instance variable *:dataLink*. An assignment to the field variable occurs within this data code block. *:getData()* reads the edit buffer of an Xbase Part and writes it into the database field via *:dataLink*.

In the example, two pushbuttons are programmed to provide navigation in the database. At any given time they can be used to select the next or previous record. This occurs in code blocks assigned to the *:activate* callback slots of the pushbuttons. In both code blocks the method *:getData()* is called before the record pointer is moved and *:setData()* is called after the DbSkip(). One of the special characteristics of an event driven GUI application is that the user controls the program execution. In the program itself it can not be predicted when a particular pushbutton will be pushed or when input in the entry field will be performed. Due to this uncertainty, the current value in the edit buffer of the entry field must be written back into the database before the record pointer is moved. And in reverse, the contents of the database field must be copied into the edit buffer immediately after the pointer is moved.

In summary, the functionality of the DataRef class is used by all Xbase Parts that can edit data (all those that have an edit buffer). These XBPs are derived from both the XbpWindow and the DataRef classes. DataRef provides the mechanism for the connection between the XBP and variables (either field variables or memory variables). To create this connection, the DataRef class uses the code block *:dataLink* and the methods *:setData()* and *:getData()*. The DataRef class also provides methods that perform data validation (the *:validate()* method) or void changes made to the edit buffer (the *:undo()* method). See the reference documentation for DataRef() or the example program in the section "Creating GUI applications" for more information.

14.3. Creating GUI applications

This section shows how to design and program GUI applications. It serves both as a guide for programmers new to a graphic user interface and as a source of solutions to some of the problems that may arise when programming GUI applications. Most of the example programs discussed in this section are also provided in the Xbase++ installation. Tips are included for organizing GUI applications and the answers to questions such as "How is this programmed?" and "Where is this implemented?" are discussed.

14.3.1. Tasks of AppSys()

The main task of the AppSys() function is to create the application window. Since AppSys() is an implicit INIT PROCEDURE, it is always called prior to the Main procedure. The application window object created in AppSys() depends on the type of application. It could be an XbpCrt window or an XbpDialog window. In order to ensure the widest possible compatibility, the default AppSys() routine included in Xbase++ creates an XbpCrt window. When developing new GUI applications using Xbase++, AppSys() should generally be changed to create an XbpDialog window instead. Additional tasks that must be performed only once at application startup can also be included in this procedure. This often includes creating the menu system, providing a help routine and initializing system wide variables or other necessary resources. These tasks can be accomplished before the application window is even visible, which allows the essential parts of the application to already be available when the Main procedure is called.

The first decision in implementing AppSys() is whether to use XbpCrt or XbpDialog windows. In the case of a GUI application, the application type must also be considered. The concept "application type" designates the kind of user interface that the application will provide. The simpler case is an SDI application (Single Document Interface) where the application consists of a single window. The alternative is an MDI application (Multiple Document Interface). An MDI application runs in multiple windows and AppSys() just creates the main window allowing the additional windows within the main window to be generated later in the program. The size of the application window generally depends on the application type. The application window of an SDI application can be smaller than that of an MDI application, since no additional windows are needed in an SDI application. Also, the window size of an SDI application can be fixed, not allowing the user to change the size. The size of the application window of an MDI application must be changeable by the user. The following example is the AppSys() procedure from the file SDIDEMO.PRG that presents a complete example of an SDI application. The various tasks performed by AppSys() are demonstrated in the following procedure:

```
PROCEDURE AppSys
   LOCAL oDlg, oXbp, aPos[2], aSize, nHeight:=400, nWidth := 615

   // Get size of desktop window
   // to center the application window
   aSize    := SetAppWindow():currentSize()
   aPos[1]  := Int( (aSize[1]-nWidth ) / 2 )
   aPos[2]  := Int( (aSize[2]-nHeight) / 2 )

   // Create application window
   oDlg := XbpDialog():new()
   oDlg:title := "Toys & Fun Inc. [Xbase++ - SDI Demo]"
   oDlg:border:= XBPDLG_THINBORDER
   oDlg:create( ,, aPos, {nWidth, nHeight},, .F. )

   // Set background color for drawing area
   oDlg:drawingArea:SetColorBG( GRA_CLR_PALEGRAY )

   // Select font
   oDlg:drawingArea:SetFontCompoundName( "8.Help.normal" )

   // Create menu system (UDF)
   MenuCreate( oDlg:menuBar() )

   // Provide online help via UDF
   oXbp := XbpHelpLabel():new():create()
   oXbp:helpObject := ;
        HelpObject( "SDIDEMO.HLP", "Help for SDI demo" )
   oDlg:helpLink := oXbp

   // Display application window and set focus to it
   oDlg:show()
   SetAppWindow( oDlg )
   SetAppFocus ( oDlg )

RETURN
```

In this example, a dialog window is created with the size 615 x 400 pixels. This size allows it to be completely displayed even on a low resolution screen. The window is provided for an SDI application and has a fixed size (XBPDLG_THINBORDER). The first call to

SetAppWindow() provides a reference to the desktop on which the application window is displayed. The *:currentSize()* method of this object provides the size of the desktop window corresponding to the current screen resolution. This information is used to position the application window when the Xbase++ application is called. In the example, the application window is displayed centered on the screen.

After the background color for the drawing area of the dialog window is set, the menu system is generated in the function MenuCreate(). This user-defined function (UDF) receives the return value of the method *:menuBar()* as an argument. The *:menuBar()* method creates an XbpMenuBar object and installs it in the application window. The menu system must then be constructed in the UDF. This approach is recommended because the menu system construction can be performed before the application window is visible. The mechanics of constructing a menu system is described in the next section.

In this AppSys() example, the mechanism for the online help is implemented after the menu system is created in MenuCreate(). This includes the generation of an XbpHelpLabel object that is assigned to the instance variable *:helpLink*. The help label object references help information and activates the window of the online help. The online help window is in turn managed by an XbpHelp object which must be provided to the XbpHelpLabel object. This is done by assigning an XbpHelp object to the *:helpObject* instance variable of the XbpHelpLabel object. The XbpHelp object manages online help windows and should exist only once within an Xbase++ application. For this reason it is created in the user-defined function HelpObject() which is shown below:

```
********************************************************************
* Routine to retrieve the help object. It manages the online help
********************************************************************
FUNCTION HelpObject( cHelpFile, cTitle )
   STATIC soHelp

   IF soHelp == NIL
      soHelp := XbpHelp():new()
      soHelp:resGeneralHelp := IPFID_HELP_GENERAL
      soHelp:resKeysHelp    := IPFID_HELP_KEYS
      soHelp:create( SetAppWindow(), cHelpFile, cTitle )
   ENDIF
RETURN soHelp
```

An XbpHelp object is created and stored in a STATIC variable the first time this function is called. The XbpHelp object manages the online help window of an Xbase++ application and a reference to it can be retrieved by calling the function HelpObject() at any point in the program. This allows any number of XbpHelpLabel objects to be created that always activate the same XbpHelp object (or the same online help).

The function HelpObject() needs to receive the file name for the HLP file and the window title for the online help. Otherwise this function is generic. It also uses two #define constants which reference the two help windows available in each application. These constants can only be user-defined and must be used in the source code of the online help as numeric IDs in order to reference the specific help window (additional information about the construction of online help is in the chapter "The IPFC help compiler").

14.3.2. The menu system of an application

The menu system of a GUI application plays a centrol role for program control. This menu must be created only once, generally within the function AppSys() prior to the first display of the application window. When the menu system is created within AppSys() or before the application window is displayed, the user does not see the construction of the menu system. The menu in the application window is already complete when the window is displayed for the first time.

The menu system consists of an XbpMenuBar object which manages the horizontal menu bar in the application window and several XbpMenu objects that are inserted in the menu bar as submenus. There are several ways to implement program control using menus. The simplest form is shown in the SDIMENU.PRG file (which presents an example SDI application). The most important steps are shown in the following code:

```
/* Call in AppSys() */

MenuCreate( oDlg:menuBar() )

********************************************************************
* Create menu system in the menu bar of the dialog
********************************************************************
PROCEDURE MenuCreate( oMenuBar )
   LOCAL oMenu

   // First sub-menu
   //
   oMenu := SubMenuNew( oMenuBar, "~File" )

   oMenu:addItem( { "Options", } )
   oMenu:addItem( MENUITEM_SEPARATOR )
   oMenu:addItem( { "~Exit" , NIL } )

   oMenu:activateItem := ;
      {|nItem,mp2,obj| MenuSelect(obj, 100+nItem) }

   oMenuBar:addItem( {oMenu, NIL} )

   // Second sub-menu  -> customer data
   //
   oMenu := SubMenuNew( oMenuBar, "C~ustomer" )
   oMenu:setName( CUST_MENU )
   oMenu:addItem( { "~New"    , NIL } )
   oMenu:addItem( { "~Seek"   , NIL } )
   oMenu:addItem( { "~Change", NIL } )
   oMenu:addItem( { "~Delete",NIL , 0, ;
                    XBPMENUBAR_MIA_DISABLED } )
   oMenu:addItem( { "~Print" ,NIL , 0, ;
                    XBPMENUBAR_MIA_DISABLED } )
   oMenu:activateItem := ;
      {|nItem,mp2,obj| MenuSelect(obj, 200+nItem) }

   oMenuBar:addItem( {oMenu, NIL} )

  /* And so forth... */
```

XbpMenu objects are created to contain the menu items. These submenus are created in the user-defined function SubMenuNew() (which is shown below), and menu items are attached to the submenus using the *:addItem()* method. A menu item is an array containing between two and four elements. In the simplest case the first element is the character string to be displayed as the menu item caption and the second element is NIL. Any character in the character string can be identified as a short-cut key by placing a tilde (~) in front of it. The second element is the code block to be executed when the menu item is selected by the user. In this example, instead of defining individual code blocks for each menu item a callback code block is defined for the entire submenu and the numeric position of the selected item and the menu object itself are passed to the selection routine MenuSelect(). In this routine a simple DO CASE...ENDCASE structure provides branching to the appropriate program module.

The second menu in the example is assigned a numeric ID (#define constant CUST_MENU) in the call to the method *:setName()*. This allows a specific XbpMenu object to be found later, since this value is found in the child list of the XbpMenuBar object which in turn is stored in the child list of the application window. The expression SetAppWindow():childFromName(CUST_MENU) would provide a reference to this XbpMenu object. This can be used to make individual menu items temporarily unavailable (or available again) if this is desired in a specific program situation.

Inserting submenus into the main menu is done using the method *:addItem()* executed by the XbpMenuBar object. The title of a menu serves as text for the menu item. In the example program this text is set for a new submenu as follows:

```
*******************************************************************
* Create sub-menu in a menu
*******************************************************************
FUNCTION SubMenuNew( oMenu, cTitle )
   LOCAL oSubMenu := XbpMenu():new( oMenu )
   oSubMenu:title := cTitle
RETURN oSubMenu:create()
```

In this function the main menu (or the immediately higher menu) is provided as the parent of the submenu. Assigning the title must occur prior to the call of the method *:create()* for correct positioning:

The default help menu

Each application should have a "Help" menu item that generally includes the same set of menu items. In the example program, this help menu is created by a separate procedure which creates the default menu items. Program control is implemented by code blocks that are passed to the menu in the method *:addItem()*. In this case the callback code block *:activateItem* is not used.

```
*********************************************************************
* Create standard help menu
*********************************************************************
PROCEDURE HelpMenu( oMenuBar )
   LOCAL oMenu := SubMenuNew( oMenuBar, "~Help" )
   oMenu:addItem( { "Help ~index", ;
                    {|| HelpObject():showHelpIndex() } } )

   oMenu:addItem( { "~General help", ;
                    {|| HelpObject():showGeneralHelp() } } )

   oMenu:addItem( { "~Using help", ;
                    {|| HelpObject():showHelp(IPFID_HELP_HELP) } } )

   oMenu:addItem( { "~Keys help", ;
                    {|| HelpObject():showKeysHelp() } } )

   oMenu:addItem( MENUITEM_SEPARATOR )

   oMenu:addItem( { "~Product information", ;
                    {|| MsgBox("Xbase++ SDI Demo") } } )

   oMenuBar:addItem( {oMenu, NIL} )
RETURN
```

The online help is managed by the XbpHelp object that is stored as a static variable in the user-defined function HelpObject(). This means it is always available when the function HelpObject() is called. Default help information can be called from the help menu by executing the XbpHelp object's methods provided for these purposes. A special method does not exist for the item "Using help". Here a #define constant is specified to the XbpHelp object that designates the numeric ID for the appropriate help window in the online help. The same ID must also be used in the IPF source code.

A dynamic menu for managing windows

In addition to the help menu that is available in both SDI and MDI applications, MDI applications have a second default menu that is used to bring different child windows of the MDI application to the front. The text in the title of each opened window appears as a menu item and selecting a menu item sets focus to the corresponding child window. This requires a dynamic approach to the menu, because the number of menu items corresponds to the number of open windows. A dynamic window menu is implemented for this purpose in the MDIMENU.PRG file (which is part of the source code for the MDIDEMO sample application). It is a good example of deriving new classes from an Xbase Part. To accomplish this, a way to easily determine the main menu of the application window (the parent window) is needed. The function AppMenu() is included in MDIDEMO.PRG for this purpose and returns the main menu of the application. There is also only one window menu per application so it can be stored in a STATIC variable. The function WinMenu() performs this task as shown in the following code:

```
*********************************************************************
* Create menu to manage open windows
*********************************************************************
FUNCTION WinMenu()
   STATIC soMenu

   IF soMenu == NIL
      soMenu := WindowMenu():new():create( AppMenu() )
   ENDIF
RETURN soMenu
```

The window menu is an instance of the class WindowMenu and receives the return value of AppMenu() as its parent. This means it is displayed as a submenu of the MDI application main menu. The user-defined class WindowMenu is derived from XbpMenu:

```
*********************************************************************
* Menu class for management of open windows
*********************************************************************
CLASS WindowMenu FROM XbpMenu
  EXPORTED:
    CLASS VAR windowStack
    CLASS METHOD initClass
    METHOD init, addItem, delItem, setItem
ENDCLASS

*********************************************************************
// Stack for open dialog windows as class variable
//
CLASS METHOD WindowMenu:initClass
   ::windowStack := {}
RETURN self
```

The class variable *:windowStack* is declared to reference opened windows. The class method *:initClass()*, whose only task is to initialize the class variable with an empty array is also included. The four methods of the XbpMenu class are overloaded. The method *:init()* is executed immediately after the class method *:new()* terminates. The *:init()* method of the XbpMenu class must also be called in order to initialize the member variables implemented there:

```
**********************************************************************
// Select a window via callback code block
//
METHOD WindowMenu:init( oParent, aPresParam, lVisible )
   ::xbpMenu:init( oParent, aPresParam, lVisible )
   ::title        := "~Window"
   ::activateItem := ;
      {|nItem,mp2,obj| SetAppFocus( obj:windowStack[nItem] ) }
RETURN self
```

After the superclass is initialized, the menu title is assigned in *:init()*. A code block is assigned to the callback slot *:activateItem.* This code block sets the focus to the window whose window title is selected from the menu. The numeric position of the selected menu item is passed to the code block as the parameter ***nItem*** and ***obj*** contains a reference to the menu object itself. Within this code block, the class variable *:windowStack* is accessed. *:windowStack* contains references to all the child windows of the MDI application. The selected window is passed to the function SetAppFocus() which sets it as the foreground window.

The last three methods of the window menu class allow menu items to be inserted, changed or deleted. These methods have the same names as methods of the XbpMenu class but the parameter passed to the methods are different. Instead of an array with between two and four elements, the passed parameter is an XbpDialog or XbpCrt object that is to receive focus if the menu item is selected.

```
**********************************************************************
// Use title of the dialog window as text for menu item
//
METHOD WindowMenu:addItem( oDlg )
   LOCAL cItem := oDlg:getTitle()

   AAdd( ::windowStack, oDlg )

   ::xbpMenu:addItem( {cItem, NIL} )
   IF ::numItems() == 1
      ::setParent():insItem( ::setParent():numItems(), {self, NIL} )
   ENDIF
RETURN self
```

An opened window is passed to the *:addItem()* method. Within this method, the window is added to the class variable *:windowStack* and the window title is added as a menu item by passing it to the :addItem() method of the XbpMenu class. A special characteristic of the window menu is that it is only displayed in the main menu when at least one child window is open. Otherwise the "Window" menu item does not appear in the main menu. The window menu inserts itself as a menu item in its parent (the main menu) after the first time the method *:addItem()* is executed.

```
*********************************************************************
// Transfer changed window title to menu item
//
METHOD WindowMenu:setItem( oDlg )
   LOCAL aItem, i := AScan( ::windowStack, oDlg )

   IF i == 0
      ::addItem( oDlg )
   ELSE
      aItem := ::xbpMenu:getItem(i)
      aItem[1] := oDlg:getTitle()
      ::xbpMenu:setItem( i, aItem )
   ENDIF

RETURN self

*********************************************************************
// Delete dialog window from window stack and from menu
//
METHOD WindowMenu:delItem( oDlg )
   LOCAL i    := AScan( ::windowStack, oDlg )
   LOCAL nPos := ::setParent():numItems()-1  // window menu is always
                                             // next to last
   IF i > 0
      ::xbpMenu:delItem( i )
      ADel( ::windowStack, i )
      Asize( ::windowStack, Len(::windowStack)-1)
      IF ::numItems() == 0
         ::setParent():delItem( nPos )
      ENDIF
   ENDIF
RETURN self
```

The *:setItem()* method is used when the window title of an opened dialog window changes. This change must also be made in the menu item of the dynamic window menu. The *:delItem()* method is called when a dialog window is closed. This method removes the title of the dialog window from the window menu. If no child windows remain open, the window menu removes itself from the main menu (the parent) and the menu item "Window" is no longer visible.

14.3.3. Tasks of the Main procedure

After the application window including the menu system has been created in AppSys(), program execution continues in the Main procedure (assuming there is no other INIT PROCEDURE). At the start of the Main procedure all conditions required for an error free run of the GUI application should be checked. For example, this might include testing for the existence of all required files, creating index files that are not available and initialization of variables required throughout the application (PUBLIC variables). Retrieving configuration variables using the command RESTORE FROM should also generally occur within the Main procedure before the program goes into the event loop. The event loop performs the central task of the Main procedure. In this loop, events are retrieved and sent on to the addressee. The following program code is from the MDIDEMO.PRG file and shows some of what needs to be included in the Main procedure or in functions called by the Main procedure. (Note: the example is not intended to cover all aspects that might be included in a Main procedure.)

```
#include "Gra.ch"
#include "Xbp.ch"
#include "AppEvent.ch"
#include "Mdidemo.ch"

*********************************************************************
* Main procedure and event loop
*********************************************************************
PROCEDURE Main
   LOCAL nEvent, mp1, mp2, oXbp
   FIELD CUSTNO, LASTNAME, FIRSTNAME, PARTNO, PARTNAME

   // Check index files and create them if not existing
   IF ! AllFilesExist( { "CUSTA.NTX", "CUSTB.NTX", ;
                         "PARTA.NTX", "PARTB.NTX"  } )
      USE Customer EXCLUSIVE
      INDEX ON CustNo                     TO CustA
      INDEX ON Upper(LastName+Firstname) TO CustB

      USE Parts EXCLUSIVE
      INDEX ON Upper(PartNo)    TO PartA
      INDEX ON Upper(PartName)  TO PartB

      CLOSE DATABASE
   ENDIF

   SET DELETED ON

   // Infinite loop. The programm is terminated in AppQuit()
   DO WHILE .T.
      nEvent := AppEvent( @mp1, @mp2, @oXbp )
      oXbp:handleEvent( nEvent, mp1, mp2 )
   ENDDO
RETURN

*********************************************************************
* Check if all files of the array 'aFiles' exist
*********************************************************************
FUNCTION AllFilesExist( aFiles )
   LOCAL lExist := .T., i:=0, imax := Len(aFiles)

   DO WHILE ++i <= imax .AND. lExist
      lExist := File( aFiles[i] )
   ENDDO
RETURN lExist
```

In this example, the Main procedure simply tests whether all the index files exist and recreates the index files if any are not found. The existence of the files is tested in the function AllFilesExist(). When this is complete, the Main procedure enters an infinite loop that reads events from the queue using AppEvent() and sends them on to the addressee by calling the addressee's method *:handleEvent()*.

Looking at this implementation, the inevitable question is: Where and how is the program terminated? The infinite loop in the Main procedure can not be terminated based on its condition DO WHILE .T.. A separate routine is used to terminate the program. The code for this routine is shown below:

```
*********************************************************************
* Routine to terminate the programm
*********************************************************************
PROCEDURE AppQuit()
   LOCAL nButton

   nButton := ConfirmBox( , ;
                "Do you really want to quit ?", ;
                "Quit", ;
                 XBPMB_YESNO , ;
                 XBPMB_QUESTION+XBPMB_APPMODAL+XBPMB_MOVEABLE )

   IF nButton == XBPMB_RET_YES
      COMMIT
      CLOSE ALL
      QUIT
   ENDIF

RETURN
```

In the termination routine AppQuit(), confirmation that the program should actually be terminated is received from the user via the ConfirmBox() function. If the application is to be terminated, all data buffers are written back into the files and all database are closed using

CLOSE ALL. The command QUIT then terminates the program. If the user does not confirm that the program should be terminated, the infinite loop in the Main procedure is continued.

It is generally recommended that the source code for a GUI application be broken down into three sections: program start, program execution and program end. The program start is contained in AppSys() and the program code executed within the Main procedure prior to the event loop. The event loop itself is the program execution. Often within this loop the program code that was generated in MenuCreate() during program start up is called by the menu system. Program termination occurs in the user defined procedure AppQuit(), where verification by the user can be requested and any data can be saved.

There are only two places in a program where the procedure AppQuit() is called. AppQuit() is generally called from a menu item and from a callback code block or from a callback method. The next two lines illustrate this:

```
oMenu:addItem( {"~Quit", {|| AppQuit() } } )
oDialog:close := {|| AppQuit() }
```

In the first line, AppQuit() is executed after a menu item is selected so there must obviously be a menu containing a menu item to terminate the application. The second line defines a callback code block for the dialog window to execute after the system menu icon of the dialog window is double clicked or the "Close" menu item is selected in the system menu of the window. Generally, the routine for terminating a GUI application should be available in the menu of the application as well as in response to the xbeP_Close event.

14.3.4. A DataDialog class for integrating databases

An important aspect in programming GUI applications is the connection between the elements of the dialog window and the DatabaseEngine. The link between a single dialog element and a single database field is created via the data code block contained in the instance variable *:dataLink* of the DataRef class that manages data. This mechanism is described in the section "DataRef() - The connection between XBP and DBE". A window generally contains several dialog elements that are linked to different database fields. Special situations can result that must be considered when programming GUI applications. The programmer must also remember that such an application is completely event driven. As soon as there is a menu system in a window, an exactly defined order of program execution is no longer assured since the user has control of the application rather than the programmer.

The two example applications SDIDEMO and MDIDEMO are provided as examples for GUI applications under Xbase^{++}. The difficulties that arise in accessing databases are taken into account in different ways in these two programs. In SDIDEMO, a procedural approach is implemented and an object oriented style is used in MDIDEMO. Both of these program examples solve the problem of non-modality of entry fields resulting from the event driven nature of a GUI application. The problems of non-modality are described by the questions: "When and how is data input validated?" and "When is data written to the database?". Since data entry fields can be activated with a mouse click, prevalidation (validation before data is entered) is not possible (after a mouse click an entry field has the input focus). This condition requires some consideration by programmers who have previously developed only under DOS without a mouse. Validating data in a GUI application can occur in the framework of postvalidation (validation after data is entered). The *:validate()* method in the DataRef class serves this purpose. If postvalidation fails, the method *:undo()* of the entry field (Xbase Part) should be called. In an event driven application, this is the only way to assure that no invalid data is written into the database.

However, the major task in programming GUI applications is generally not validating the data, but transferring the input data to the database. In the SDIDEMO and MDIDEMO example programs, the philosophy is used that the data needs to be written to the database when the record pointer is changed. All Xbase Parts have their own edit buffer to hold the modified data and the value to write into the database fields is stored in this edit buffer of each Xbase Part. For all of the database fields that can be changed within a dialog window, an Xbase Part must exist to store the value in its edit buffer. The following code fragment illustrates this:

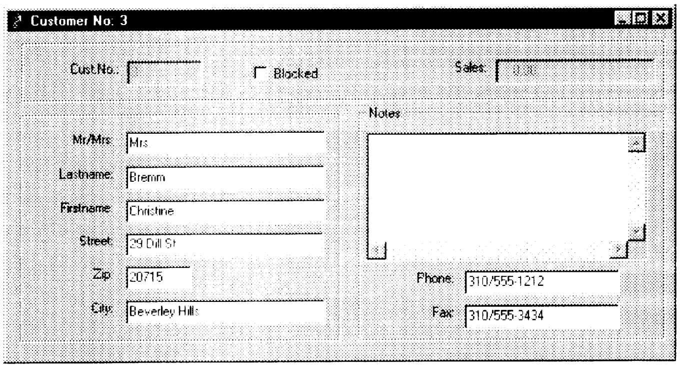
```
oXbp := XbpSLE():new( oDlg:drawingArea,, {95,135}, {180,22} )
oXbp:bufferLength := 20
oXbp:dataLink := {|x| IIf( x==NIL, LASTNAME, LASTNAME := x ) }
oXbp:create():setData()
```

In this code, an entry field is created for editing the data in the database field LASTNAME. Calling the method *:setdata()* in connection with *:create()* copies the data from the database field into the edit buffer of the XbpSLE object. Within a dialog window any number of entry fields can exist to access database fields. The edit buffer of all entry fields in the dialog window can be changed at any time (a mouse click in an entry field is sufficient to begin editing). For this reason, it must be determined when changes to the data in an entry field will be copied back into the file. There are two approaches: changes to individual data entry fields are written into the file as soon as the change occurs or all changes from all data entry fields in a window are written into the file as soon as a "Save" routine is explicitly called or the record pointer is repositioned.

The second approach is preferred in GUI applications that are designed for simultaneous access on a network. This approach allows several data entry fields to be changed in a dialog window without each change being individually copied to the database. In concurrent or network operation saving each change to the database would require a time consuming lock and release of the current record. A performance optimized GUI application only locks a record when it can write several fields to the database or when the record pointer changes.

The problems of validating and saving data into databases is present in every application. The following code shows several aspects of this problem and is based on the example application MDIDEMO. In this example application the DataDialog class is used to provide dialog windows for accessing the DatabaseEngine. A DataDialog object coordinates a DatabaseEngine with a dialog window. The source code for this class is contained in the file DATADLG.PRG.

An example of an input screen based on DataDialog, is shown in the following illustration:



Customer No: 3

Cust.No.:

Blocked

Sales:

Notes

Mr/Mrs: Mrs

Lastname: Bremm

Firstname: Christine

Street: 29 Dill St

Zip: 20715

City: Beverley Hills

Phone: 310/555-1212

Fax: 310/555-3434

Input screen for customer data

The DataDialog class is derived from XbpDialog. It adds seven new instance variables and eleven additional methods for transferring data from a database to the dialog and vice versa. Three of the instance variables are for internal use only and are declared as PROTECTED:. The four methods *:init()*, *:create()*, *:configure()* and *:destroy()* perform steps in the "life cycle" of a DataDialog object:

```
#include "Gra.ch"
#include "Xbp.ch"
#include "Dmlb.ch"
#include "Common.ch"
#include "Appevent.ch"

*********************************************************************
* Class declaration
*********************************************************************
CLASS DataDialog FROM XbpDialog
   PROTECTED:
     VAR appendMode            // Is it a new record?
     VAR editControls          // List of XBPs for editing data
     VAR appendControls        // List of XBPs enabled only
                               // during APPEND

   EXPORTED:
     VAR area      READONLY    // current work area
     VAR newTitle              // code block to change window title
     VAR contextMenu           // context menu for data dialog
     VAR windowMenu            // dynamic window menu in
                               // application window
```

```
      METHOD init               // overloaded methods
      METHOD create
      METHOD configure
      METHOD destroy
      METHOD addEditControl     // register XBP for edit
      METHOD addAppendControl   // register XBP for append
      METHOD notify             // process DBO message
      METHOD readData           // read data from DBF
      METHOD validateAll        // validate all data stored in XBPs
      METHOD writeData          // write data from XBPs to DBF
      METHOD isIndexUnique      // check index value for uniqueness
ENDCLASS
```

The protected instance variable *:appendMode* contains the logical value .T. (true) only when the phantom data record (record number LastRec()+1) is current. The other two protected instance variables *:editControls* and *:appendControls* are arrays containing lists of Xbase Parts that can modify data. In order to create a data dialog, editable XBPs are required as well as Xbase Parts that can not edit data but display static text or boxes (XbpStatic objects). The instance variable *:editControls* contains a list of references to those XBPs in the child list (all XBPs that are displayed in the dialog window are contained in this list) that can be edited.

The task of the *:appendControls* instance variable is similar and contains a list of XBPs that are only enabled when a new record is appended. In all other cases, these XBPs are disabled. They only display data and do not allow the data in them to be edited. This is useful for editing database fields that are contained in the primary database key which should not be changed once they are entered in the database. *:editControls* and *:appendControls* are both initialized with empty arrays. This is done in the *:init()* method after it calls the *:init()* method of the XbpDialog class as shown below:

```
******************************************************************
* Initialize data dialog
******************************************************************
METHOD DataDialog:init( oParent, oOwner , ;
                        aPos    , aSize  , ;
                        aPParam, lVisible )

   DEFAULT lVisible TO .F.

   ::xbpDialog:init( oParent, oOwner, ;
                     aPos    , aSize , ;
                     aPParam, lVisible )

   ::area           := 0
   ::border         := XBPDLG_THINBORDER
   ::maxButton      := .F.
   ::editControls   := {}
   ::appendControls := {}
   ::appendMode       := .F.
   ::newTitle         := {|obj| obj:getTitle() }

RETURN self
```

All instance variables are set to values with the valid data type in the *:init()* method. Only the instance variables *:border* and *:maxButton* change the default values assigned in the XbpDialog class. The window of a DataDialog object is fixed in size and can not be enlarged. The method has the same parameter list as the method *:new()* and *:init()* in the XbpDialog class. This allows it to receive parameters and simply pass them on to the superclass. The DataDialog is different in that it is created as hidden by default. This is recommended when many XBPs will be displayed in the window after the window is generated. The construction of the screen with the method *:show()* is faster if everything can be displayed at once after the XBPs have been added to the dialog window.

The instance variable *:newTitle* must contain a code block that the DataDialog object is passed to. For this reason a code block is defined in the *:init()* method, but it must be redefined later. This code block changes the window title while the dialog window is visible. The default code block is assigned to the instance variable in the *:init()* method to ensure that the instance variable has the correct data type.

The next method in the "life cycle" of a DataDialog object is *:create()*. A database must be open in the current work area prior to this method being called. A DataDialog object continues to use the work area that is current when the *:create()* method is executed:

```
********************************************************************
* Load system resources
* Register DataDialog in current work area
********************************************************************
METHOD DataDialog:create( oParent, oOwner , ;
                          aPos    , aSize  , ;
                          aPParam, lVisible )

   ::xbpDialog:create( oParent, oOwner , ;
                       aPos    , aSize  , ;
                       aPParam, lVisible )

   ::drawingArea:setColorBG( GRA_CLR_PALEGRAY )

   ::appendMode       := Eof()
   ::area             := Select()

   ::close            := {|mp1,mp2,obj| obj:destroy() }
   ::setDisplayFocus  := {|mp1,mp2,obj| ;
                          DbSelectArea( obj:area ) }

   DbRegisterClient( self )
```

```
RETURN self
```

The most important task of *:create()* is requesting system resources for the dialog window. This occurs when the method of the same name in the superclass is called and the parameters are simply passed on to it. The background color for the drawing area (*:drawingArea*) of the dialog window is then set. The call to *:setColorBG()* also defines the background color for all XBPs later displayed in the dialog window. This affects all XBPs that have a caption for displaying text. This simplifies programming because the background color of the individual XBPs with captions do not have to be set separately. Generally when the system colors defined in the system configuration are to be used *:setColorBG()* can not be called.

The lines that follow are important because they link the DataDialog object and the work area. First, whether the pointer is currently at Eof() is determined, then Select() determines the number of the current work area. Two code blocks are assigned to the callback slots *:close* and *:setDisplayFocus*. The method *:destroy()* (described below) is called after the xbeP_Close event. As soon as the DataDialog object receives focus, the code block in *:setDisplayFocus* is executed. In this code block, the work area managed by the DataDialog object is selected as the current work area using DbSelectArea(). This means that if the mouse is clicked in a DataDialog window, the correct work area is automatically selected.

The call to DbRegisterClient() is critical for the program logic. This registers the DataDialog object in the work area so that it is automatically notified whenever anything in the work area changes. This includes notification of changes in the position of the record pointer. When the record pointer changes, the new data must be displayed by the XBPs that are listed in the instance variable *:editControls*. This is done using the method *:notify()* which is described later after the remaining methods in the DataDialog "life cycle" are discussed. The method *:configure()* is provided to handle changes in the work area managed by the DataDialog object and is shown below:

```
*******************************************************************
* Configure system resources
* Register data dialog in new work area if necessary
*******************************************************************
METHOD DataDialog:configure( oParent, oOwner , ;
                             aPos    , aSize  , ;
                             aPParam, lVisible )
   LOCAL lRegister := (::area <> Select())

   ::xbpDialog:configure( oParent, oOwner , ;
                          aPos    , aSize  , ;
                          aPParam, lVisible )
   IF lRegister
     (::area)->( DbDeRegisterClient( self ) )
   ENDIF

   ::area       := Select()
```

```
    ::appendMode := Eof()

    IF lRegister
       DbRegisterClient( self )
    ENDIF

RETURN self
```

A DataDialog object always manipulates the current work area. Because of this, the method *:configure()* compares the instance variable *:area* to Select() to determine whether the current area has changed. If it has changed, the object is deregistered in the old work area and registered in the new area. In addition, the system resources for the dialog window are also reconfigured in the call to the *:configure()* method of the superclass.

The final method of the DataDialog life cycle is *:destroy()*. This method closes the database used by the DataDialog object and releases the system resources. The instance variables declared in the DataDialog class are reset to the values assigned in the method *:init()*:

```
*******************************************************************
* Release system resources and unregister data dialog from work area
*******************************************************************
METHOD DataDialog:destroy()

   ::writeData()
   ::hide()

   (::area)->( DbCloseArea() )

   IF ! Empty( ::windowMenu )
      ::windowMenu:delItem( self ) // delete menu item in window menu
      ::windowMenu := NIL
   ENDIF

   IF ! Empty( ::contextMenu )
      ::contextMenu:cargo := NIL   // Delete reference of data
      ::contextMenu := NIL         // dialog and context menu
   ENDIF

   ::xbpDialog:destroy()           // release system resources
   ::Area           := 0           // and set instance variables
   ::appendMode     := .F.         // to values corresponding to
   ::editControls   := {}          // :init() state
   ::appendControls := {}
   ::newTitle       := {|obj| obj:getTitle() }

RETURN self
```

The method *:writeData()* is called in *:destroy()* in order to write all the data changes into the database before it is closed using DbCloseArea(). After the database is closed, the DataDialog object is implicitly deregistered from the work area and a call to DbDeRegisterClient() is not necessary. If a menu object is contained in the instance variable *:windowMenu*, the DataDialog object is removed from the list of menu items in this menu (the WindowMenu class is described in a previous section). The instance variable *:contextMenu* can contain a context menu that is activated by clicking the right mouse button. This mechanism is described in a later section. It is essential that the reference to the DataDialog object in the instance variable *:cargo* of the context menu be deleted because the method *:destroy()* is expected to eliminate all references to the DataDialog object. If a DataDialog object remains referenced anywhere, whether in a variable, an array, or an instance variable, it will not be removed from memory by the garbage collector. This concludes the discussion of the methods that perform tasks in the "life cycle" of a DataDialog object.

One of the most important method of the DataDialog class is the *:notify()* method. This method is called whenever something is changed in the work area associated with the object. An abbreviated version of this method highlighting its essential elements is shown below:

```
*********************************************************************
* Notify method:
*   - Write data to fields prior to moving the record pointer
*   - Read data from fields after moving the record pointer
*********************************************************************
METHOD DataDialog:notify( nEvent, mp1, mp2 )

   IF nEvent <> xbeDBO_Notify            // no notify message
      RETURN self                        // ** return **
   ENDIF

   DO CASE
   CASE mp1 == DBO_MOVE_PROLOG           // record pointer is about
      ::writeData()                      // to be moved

   CASE mp1 == DBO_MOVE_DONE .OR. ;      // skip is done
        mp1 == DBO_GOBOTTOM  .OR. ;
        mp1 == DBO_GOTOP
      ::readData()

   ENDCASE
RETURN self
```

Calling the function DbRegisterClient() in the *:create()* method of the DataDialog object registers the object in the work area it uses. As soon as anything changes in this work area, the *:notify()* method is called. For record pointer movement, this method is called twice. The first time the DataDialog object receives the value represented by the constant DBO_MOVE_PROLOG (defined in the DMLB.CH file) as the *mp1* parameter. This is a signal that means "Warning the record pointer position is about to change." When it receives this message, the DataDialog object executes the method *:writeData()* which writes the data of the current record into the database. In the second call to *:notify()*, the object receives the value of the constant DBO_MOVE_DONE. This message tells the object "Ok, the pointer has been changed." In response to this message, the object executes the *:readData()* method which copies the fields of the new record into the edit buffers of the XBPs that are in the data dialog's *:editControls* instance variable. This allows the data in the new record to be edited.

The *:notify()* method provides important program logic for the DataDialog object. In this method, the DataDialog object reacts to messages sent by the work area it uses. This method is only called after the object is registered in the work area using DbRegisterClient(). Or more precisely, it is only called when the object is registered in the database object (DBO) that manages the work area (a DBO is automatically created when a database is opened). Based on the event passed, the *:notify()* event determines whether a record should be read into XBPs or whether the data in the XBPs should be written into the database. The DataDialog object does not directly manage the data but does manage the XBPs contained in the array *:editControls*. Adding XBPs to this array is done using the method *:addEditControl()*.

```
********************************************************************
* Add an edit control to internal list
********************************************************************
METHOD DataDialog:addEditControl( oXbp )
   IF AScan( ::editControls, oXbp ) == 0
      AAdd(  ::editControls, oXbp )
   ENDIF
RETURN self

********************************************************************
* Add an append control to internal list
********************************************************************
METHOD DataDialog:addAppendControl( oXbp )
   IF AScan( ::appendControls, oXbp ) == 0
      AAdd(  ::appendControls, oXbp )
   ENDIF
RETURN self
```

The two methods *:addEditControl()* and *:addAppendControl()* are almost identical. One adds an Xbase Part to an array stored in the instance variable *:editControls* and the other adds an Xbase Part to *:appendControls*. When a DataDialog object executes the method *:readData()* or *:writeData()*, it sequentially processes the elements in the *:editControls* array and sends each element (each Xbase Part) the message to read or write its data. A code fragment is included below to illustrate how Xbase Parts can be added to the window of a DataDialog object and to the *:editControls* instance variable if appropriate. The variable *oDlg* references a DataDialog object.

```
oXbp := XbpStatic():new( oDlg:drawingArea,, {5,135}, {80,22} )
oXbp:caption := "Lastname:"          // static text is stored
oXbp:options := XBPSTATIC_TEXT_RIGHT // only in the child list
oXbp:create( )

oXbp := XbpSLE():new( oDlg:drawingArea,, {95,135}, {180,22} )
oXbp:bufferLength := 20              // entry field linked to
oXbp:tabStop  := .T.                 // database
oXbp:dataLink := {|x| IIf( x==NIL, LASTNAME, LASTNAME := x ) }
oXbp:create():setData()

oDlg:addEditControl( oXbp )          // adds new XBP to :editControls
```

The Xbase Parts appear in the drawing area of a dialog window, so *oDlg:drawingArea* must be specified as the parent. The code fragment creates an XbpStatic object to display the text "Lastname:" and an XbpSLE object to access and edit the database field called LASTNAME. Passing the XbpSLE object to the method *:addEditControl()* adds this Xbase Part to the *:editControls* array. In the child list of the DataDialog object there are now two XBPs but the *:editControls* array contains only the XBP for data that can be edited. The methods *:readData()*, *:validateAll()* and *:writeData()* assume that all the Xbase Parts that can edit data are included in the *:editControls* array. The program code for *:readData()* is shown below:

```
********************************************************************
* Read current record and transfer data to edit controls
********************************************************************
METHOD DataDialog:readData()
   LOCAL i, imax  := Len( ::editControls )

   FOR i:=1 TO imax                          // Transfer data from file
      ::editControls[i]:setData()            // to XBPs
   NEXT

   Eval( ::newTitle, self )                  // Set new window title

   IF Eof()                                  // enable/disable XBPs
      IF ! ::appendMode                      // active only during
         imax  := Len( ::appendControls )    // APPEND
         FOR i:=1 TO imax                    //
            ::appendControls[i]:enable()     // Hit Eof(), so
         NEXT                                // enable XBPs
      ENDIF
      ::appendMode := .T.
   ELSEIF ::appendMode                       // Record pointer was
      imax  := Len( ::appendControls )       // moved from Eof() to
      FOR i:=1 TO imax                       // an existing record.
         ::appendControls[i]:disable()       // Disable append-only
      NEXT                                   // XBPs
      ::appendMode := .F.
   ENDIF

RETURN
```

The *:setData()* method in the first FOR...NEXT loop causes all the XBPs referenced in the instance variable *:editControls* to re-read their edit buffers by copying the return value of the data code block contained in *:dataLink* into their edit buffer. The remaining code just enables and disables the XBPs in the *:appendControls* list. In addition to reading the data in the database fields, this method is the appropriate place to enable or disable those Xbase Parts that should only be edited when a new record is being appended.

The counterpart of *:readData()* is the *:writeData()* method. In this method, the data in the edit buffer of each Xbase Part listed in *:editControls* is written back to the database. This method involves relatively extensive program code, because it performs record locking and identifies whether a new record should be appended.

```
*********************************************************************
* Write data from edit controls to file
*********************************************************************
METHOD DataDialog:writeData()
   LOCAL i, imax
   LOCAL lLocked   := .F. , ;        // Is record locked?
         lAppend   := .F. , ;        // Is record new?
         aChanged  := {}  , ;        // XBPs containing changed data
         nOldArea  := Select()       // Current work area

   dbSelectArea( ::area )

   IF Eof()                          // Append a new record
      IF ::validateAll()             // Validate data first
         APPEND BLANK
         lAppend  := .T.
         aChanged := ::editControls  // Test all possible changes
         lLocked  := ! NetErr()      // Implicit lock
      ELSE
         MsgBox("Invalid data")      // Do not write invalid data
         DbSelectArea( nOldArea )    // to new record
         RETURN .F.                  // *** RETURN ***
      ENDIF
   ELSE
      imax := Len( ::editControls ) // Find all XBPs containing
      FOR i:=1 TO imax               // changed data
         IF ::editControls[i]:changed
            AAdd( aChanged, ::editControls[i] )
         ENDIF
      NEXT

      IF Empty( aChanged )           // Nothing has changed, so
```

```
         DbSelectArea( nOldArea )    // no record lock necessary
         RETURN .T.                  // *** RETURN ***
      ENDIF

      lLocked := DbRLock( Recno() ) // Lock current record
   ENDIF

   IF ! lLocked
      MsgBox( "Record is currently locked" )
      DbSelectArea( nOldArea )       // Record lock failed
      RETURN .F.                     // *** RETURN ***
   ENDIF

   imax := Len( aChanged )           // Write access is necessary
   FOR i:=1 TO imax                  // only for changed data
      IF ! lAppend
         IF ! aChanged[i]:validate()
            aChanged[i]:undo()       // invalid data !
            LOOP                     // undo changes and validate
         ENDIF                       // next XBP
      ENDIF
      aChanged[i]:getData()          // Get data from XBP and
   NEXT                              // write to file

   DbCommit()                        // Commit file buffers
   DbRUnlock( Recno() )              // Release record lock

   IF ::appendMode                   // Disable append-only XBPs
      imax  := Len( ::appendControls ) // after APPEND
      FOR i:=1 TO imax
         ::appendControls[i]:disable()
      NEXT
      ::appendMode := .F.

      IF ! Empty( ::contextMenu )
         ::contextMenu:disableBottom()
         ::contextMenu:enableEof()
      ENDIF
   ENDIF

   DbSelectArea( nOldArea )

RETURN .T.
```

Appending a new record requires special logic in the *:writeData()* method of the DataDialog object. A special empty record (the phantom record) is automatically available when the pointer is positioned at Eof(). If the pointer is positioned at Eof(), the method *:readData()* has copied "empty" values from the database fields into the XBP edit buffers for all of the

XBPs listed in the instance variable *:editControls*. Because of this, all XBPs contain valid data types. But there is no guarantee that valid data is also contained in the edit buffers of each XBP. This means that data validation must be performed before the record is even appended. Since there is not yet a record, all the data to be saved is found only in the edit buffer of the corresponding Xbase Parts. The method *:validateAll()* is called before a new record is appended which is to receive data from the edit buffers of the Xbase Parts.

Data validation is especially important when a new record is appended because no previously valid data exists to allow the changes to the individual edit buffers to be voided. For records that are being edited, the method *:undo()* allows changes to the values in the edit buffers to be voided. But this approach assumes there is an original field value that is valid. This is only true if the record being edited existed prior to editing. When a record is appended, the original values are "empty" values which are probably not valid.

In *:writeData()*, this situation is handled by calling the method *:validateAll()* before the new record is appended to the file using APPEND BLANK. If data validation fails on even one field, a message box containing the text "Invalid data" is displayed and a new record is not appended. The invalid data remains in the edit buffers of the corresponding Xbase Parts (*:editControls*) and can be corrected by the user. When an existing record is edited, data validation occurs individually for each Xbase Part. If the *:validate()* method of an XBP returns the value .F. (false) (indicating invalid data), the *:undo()* method of the XBP is executed which copies the original, valid data back into the edit buffer.

If any XBPs listed in *:editControls* have been changed, the record is locked using DbRLock(). After validation, data from the edit buffers is written into the database by calling the method *:getData()*. The function DbCommit() ensures that data in the file buffers are written into the database. Finally the record lock is released.

The *:writeData()* method handles the problems of data validation and appending records as they occur in an event driven environment. This process is controlled by the mouse or rather by the user who causes the mouse clicks. Even though *:writeData()* is called from only one place in the *:notify()* method, it is impossible to foresee when this method will be called. While it is clear that it is called when the record pointer moves it is not possible to predict which record will be current when the method *:writeData()* is called. The special case occurs when the pointer is located on the phantom record. In this case data validation cannot be reversed using the *:undo()* method because no previously validated data exists. For this reason, all data must be validated before a new record can be appended. The method which checks that all data is valid is called *:validateAll()* and is shown below:

```
*********************************************************************
* Validate data of all edit controls
* This is necessary prior to appending a new record to the database
*********************************************************************
METHOD DataDialog:validateAll()
   LOCAL i := 0, imax := Len( ::editControls )
   LOCAL lValid := .T.

   DO WHILE ++i <= imax .AND. lValid
      lValid := ::editControls[i]:validate()
   ENDDO

RETURN lValid
```

The method consists only of a DO WHILE loop that is terminated as soon as the XBP :validate method signals invalid data. The method *:validate()* is executed for all XBPs listed in *:editControls*. This method always returns the value .T. (true) unless there is a code block contained in the instance variable *:validate*. If a code block is contained in this instance variable, it is executed and receives the XBP as the first parameter. This code block performs data validation and returns .T. (true) if the data is valid.

Special data validation is needed for the primary key in a database. The primary key is the value in the database that uniquely identifies each record. There is always an index for the primary key. The method *:isIndexUnique()* (shown below) tests whether a value already exists as a primary key in an index file of the database. This method demonstrates an extremely important aspect for the use of DataDialog objects (more precisely: for the use of the function DbRegisterClient()):

```
********************************************************************
* Check whether an index value does *not* exist in an index
********************************************************************
METHOD DataDialog:isIndexUnique( nOrder, xIndexValue )
   LOCAL nOldOrder := OrdNumber()
   LOCAL nRecno    := Recno()
   LOCAL lUnique   := .F.

   DbDeRegisterClient( self )        // Suppress notification from DBO
                                     // to self during DbSeek() !!!
   OrdSetFocus( nOrder )

   lUnique := .NOT. DbSeek( xIndexValue )

   OrdSetFocus( nOldOrder )
   DbGoTo( nRecno )

   DbRegisterClient( self )

RETURN lUnique
```

The functionality of the *:isIndexUnique()* method is very limited. All it does is search for a value in the specified index and return the value .T. (true) if the value is not found. An important point shown here is that the DataDialog object executing the method must be deregistered in the work area. It was initially registered in the work area by the method

:create(), causing the method *:notify()* to be called every time the record pointer changes. In this case, it is a method of the DataDialog object changing the pointer by calling DbSeek(). If the DataDialog object were not deregistered, an implicit recursion would result since each change to the pointer via DbSeek() calls the method *:notify()*. For this reason, DbDeRegisterClient() is used to deregister the DataDialog object prior to the call to DbSeek(). It is again registered in the work area using DbRegisterClient() after DBSeek().

In summary, the DataDialog class solves many problems which must be considered when programming GUI applications that work with databases. Record pointer movements are easily identified in the method *:notify()* that is automatically called when the DataDialog object is registered in the current work area using the function DbRegisterClient(). Before the record pointer is moved, a DataDialog object copies the changed data in *:editControls* back into the database. After the record pointer is changed, a DataDialog object displays the current data. Data validation occurs prior to data being written into the database either by a new record being appended or existing data being overwritten. Whether new data is being saved or existing data modified is determined by the DataDialog object.

14.3.5. DataDialog and data entry screens

Objects of the DataDialog class described in the previous section are appropriate for programming data entry screens in GUI applications. Each input screen is an independent window that is displayed as a child of the application window. In each child window (input screen) Xbase Parts are added to edit the database fields. Because they are separate windows, it is recommended that each entry screen be programmed in a separate routine. The tasks of this routine include opening all databases required for the entry screen, creating the child window (DataDialog), and adding the Xbase Parts needed for editing the database fields to the entry screen. In the example application MDIDEMO, two entry screens are programmed, one for customer data and one for parts data. The process of creating the data entry screen is the same in both cases. Sections of the program code from the file MDICUST.PRG are discussed below to illustrate various aspects significant when programming data entry screens:

```
*********************************************************************
* Customer Dialog
*********************************************************************
PROCEDURE Customer( nRecno )
   LOCAL oXbp, oStatic, drawingArea, oDlg
   FIELD CUSTNO, MR_MRS, LASTNAME, FIRSTNAME, STREET, CITY, ZIP , ;
         PHONE , FAX    , NOTES    , BLOCKED   , TOTALSALES

   IF ! OpenCustomer( nRecno )        // open customer database
      RETURN
   ENDIF
   oDlg := DataDialog():new( RootWindow():drawingArea ,, ;
                             {100,100}, {605,315},, .F.  )
   oDlg:title := "Customer No: "+ LTrim( CUSTNO )
   oDlg:icon  := ICON_CUSTOMER
   oDlg:create()
   /* ... */
```

The Customer() procedure creates a new child window in the MDI application where customer data can be edited. LOCAL variables are first declared to reference the Xbase Parts created and all of the database fields are identified to the compiler as field variables. Before the child window (DataDialog) is created, the required database(s) must be open. This occurs in the function OpenCustomer() which returns the value .F. (false) only if the customer database could not be opened. Opening the database might fail because another workstation has the file exclusively open or the file is simply not found. (Note: testing for the existence of files should have already been done at startup in the Main procedure).

When the required file(s) can be opened, the dialog window is created. This is done using DataDialog class method *:new()* which generates a new instance of the DataDialog class. The parent of the new object is the drawing area (*:drawingArea*) of the application window created in AppSys() and returned by the user-defined function RootWindow(). As soon as the child window is created, the Resource ID for an icon must be entered into the instance variable *:icon*. This icon is displayed within the application window when the child window is minimized. In this example, the #define constant ICON_CUSTOMER is used. An icon is declared in a resource file and must be linked to the executable file using the resource compiler ARC.EXE. If no icon ID is specified for a child window, the window contents in the range from point {0,0} to point {32,32} are used as the icon when the window is minimized. This means everything visible in the lower left corner of the child window up to the point {32,32} appears as the symbol for the minimized child window.

In the example program MDICUST.PRG, only the CUSTOMER.DBF database file needs to be opened. It is important for the customer database to be reopened each time the procedure Customer() is called. This is shown in the program code of the function OpenCustomer():

```
********************************************************************
* Open customer database
********************************************************************
FUNCTION OpenCustomer( nRecno )
   LOCAL nOldArea := Select(), lDone := .F.

   USE Customer NEW
   IF ! NetErr()
      SET INDEX TO CustA, CustB
      IF nRecno <> NIL
         DbGoto( nRecno )
      ENDIF
      lDone := .T.
   ELSE
```

```
      DbSelectArea( nOldArea )
      MsgBox( "Database can not be opened" )
   ENDIF

RETURN lDone
```

Each instance of the DataDialog class (each data entry screen) manages its own work area. If the Customer() procedure is executed 10 times, 10 data entry screens are created for customer data and the CUSTOMER.DBF file is opened 10 times. This rule is standard for event oriented GUI applications: **Each dialog opens its own database**. This requires some consideration for programmers coming from DOS procedural programming, since the same approach is not appropriate under DOS because of the 255 file handle limit. This limit does not exist under a 32bit operating system. Access to a single database file from several dialogs does require that protection mechanisms be implemented in the Xbase^{++} application. The mechanisms for locking records or files is sufficient under Xbase^{++} so that when the program allows simultaneous access on a network, it will also handle the file being opened multiple times within a single application.

Each call to OpenCustomer() opens the customer database in a new work area and the method *DataDialog:create()* registers the dialog window in this new work area. From this point on, the DataDialog object is notified about each change in the work area (via the method *:notify()*) and can be assigned the appropriate XBPs (via *:editControls*) so that they can automatically be handled by the DataDialog methods. (Note for Clipper programmers: the expression USE Customer NEW is allowed in Xbase^{++} without specifying an alias name. If a database is opened multiple times, Xbase^{++} provides a unique alias name formed from the file name and the number of the current work area).

When the database is open and the DataDialog (the child window) is created, the most important processes for programming a data entry screen are nearly complete. Xbase Parts must still be added to the dialog window. These include both XBPs that contribute to the visual organization of the data entry screen (borders and text) and XBPs that allow access to database fields via *:dataLink*. This second group is primarily made up of objects from the classes XbpSLE, XbpCheckBox and XbpMLE. XbpSLE objects provide single line data entry fields, XbpCheckBox objects manage logical values and XbpMLE objects provide multiple line data entry fields that allow memo fields to be edited. Objects of the classes XbpSLE, XbpCheckBox and XbpMLE are sufficient to program the sections of data entry screens where database fields are edited.

Boxes that are displayed by XbpStatic objects are used to provide visually organization of data entry screens. Entry fields are not only visual separated when they appear in a box, but the fields can also be grouped in the program logic. The following program section shows another example of code defining a part of a data entry screen.

This example is a continuation of the Customer() procedure:

```
   // Get drawing area from dialog
   drawingArea := oDlg:drawingArea

   oStatic := XbpStatic():new( drawingArea ,, {12,227}, {579,58} )
   oStatic:type := XBPSTATIC_TYPE_GROUPBOX
   oStatic:create()
```

In the above sample, the drawing area (*:drawingArea*) of the dialog window is retrieved and passed as the parent for the dialog elements to be displayed in the window. The first dialog element is an XbpStatic object responsible for displaying a group box. This box displays text (the caption) in the upper left corner. A group box is used for grouping data entry fields and acts as the parent for all the Xbase Parts which are displayed within the group box. In other words: the parent for a group box is the *:drawingArea* and the parent for the Xbase-Parts displayed within the group box is the XbpStatic object representing the box. For this reason the XbpStatic object is referenced in the variable *oStatic* and is used as the parent in the example of creating data entry fields shown below:

```
oXbp := XbpSLE():new( oStatic,, {95,135}, {180,22} )
oXbp:bufferLength := 20
oXbp:dataLink := {|x| IIf( x==NIL, Trim(LASTNAME), LASTNAME := x ) }
oXbp:create():setData()

oDlg:addEditControl( oXbp )   // register Xbp as EditControl
```

In the above sample, a data entry field is created for display within a group box (the parent of the data entry field is oStatic). The XbpSLE object accesses the database field NAME and the length of the edit buffer is limited to the length of the database field. The field LASTNAME has 20 characters in the example. A general incompatibility between database fields and XbpSLE objects is handled in the *:dataLink* code block. When data is read from the database field, the padding blank spaces are included. If the data is copied directly from the field LASTNAME into the edit buffer of the XbpSLE object, 20 characters are always included in the edit buffer even for a name such as "Smith" that is only five characters long. The blank spaces stored in the database field are copied into the edit buffer of the XbpSLE object. The result is that the edit buffer of the XBP object is already full and characters can only be added to the edit buffer in "Overwrite" mode. An XbpSLE object considers blank spaces as fully valid characters and to prevent these problems, the blank spaces at the end of the name (trailing spaces) are explicitly removed using Trim() when the data is read from the database field LASTNAME within *:dataLink.*

An XbpSLE object can only edit values in its edit buffer that are of "character" type. The maximum number of characters is 32KB. Values of numeric or date type must be converted to a character string when copied into the edit buffer of an XbpSLE object and converted back to the correct type before being written into the database field. This must be done in the data code block contained in *:dataLink.*

Examples for code blocks which perform type conversions are shown below:

```
oXbp:dataLink := {|x| IIf(x==NIL, Transform( FIELD->NUMERIC, "@N"), ;
                                  FIELD->NUMERIC := Val(x) ) }
oXbp:dataLink := {|x| IIf(x==NIL, DtoC( FIELD->DATE ), ;
                                  FIELD->DATE := CtoD(x) ) }
```

When database fields are read into the edit buffer of an XbpSLE object blank spaces must be deleted and numeric and date values must be converted to character strings. When the modified data is saved to the database fields, values for date and numeric fields must again be converted to the correct data type. This task is performed by the data code block assigned to the instance variable *:dataLink*.

Another task of the data code block contained in *:dataLink* occurs when more than one file is required for the data entry screen (the DataDialog). In this case, fields from several databases are edited in a single data entry screen and the data code block must also select the correct work area for the field variable.

14.3.6. Program control in dialog windows

The previous discussions of GUI application concepts have focused on the basic organization of a GUI program. The key issues discussed were the program start, program execution, and program end. These correspond to AppSys() with a menu system, the Main procedure with the event loop and AppQuit(), respectively. The DataDialog class was discussed as a mechanism for linking dialog windows with DatabaseEngines. This class offers solutions to problems that can occur during simultaneous access on a network or when a database is opened multiple times in a single application. In the previous section, incorporating Xbase Parts into a dialog window was illustrated. The final remaining question for programming GUI applications is: How is the program controlled within an individual dialog window?

A distinction must be made between controlling a window and controlling an application. The overall running of the application is controlled by the application menu installed in the application window. In an SDI application, control of the application is basically the same as control of the dialog window, since the application consists of only a single dialog window. In the SDIDEMO example application, control of the application through the menu system includes selecting the data entry screens for customer data or for parts data. Control within windows occurs using pushbuttons that allow record pointer movement within the customer file or parts file, cause the current data to be saved or terminate the data input.

In the example application MDIDEMO, the application control is limited to opening the customer or parts data entry screen. A child window presents data for a customer or a part. As soon as a child window is opened, the application and the application menu no longer have control over the newly opened window. Program control within a child window is performed in an MDI application by a context menu that is an essential control element for program control. A context menu is generally activated by clicking the right mouse button. It is displayed on the screen as a Popup menu. Its menu items provide a selection of actions that are appropriate to execute within the window or in relation to the dialog element where the right mouse click occurred.

The context menu in the MDIDEMO example application includes program control of database navigation (DbSkip(), DbGoBottom(), DbGoTop()) and elementary database operations such as "Search", "Delete" and "New record". Programming a context menu requires the definition of an XbpMenu object and is otherwise similar to programming application menu objects. As an example, the program code to create the context menu for the customer database used in MDIDEMO is shown below:

```
********************************************************************
* Create context menu for customer dialog
********************************************************************
STATIC FUNCTION ContextMenu()
   STATIC soMenu

   IF soMenu == NIL
      soMenu       := DataDialogMenu():new()
      soMenu:title := "Customer context menu"
      soMenu:create()

      soMenu:addItem( { "~New", ;
                        {|mp1,mp2,obj| DbGoTo( LastRec()+1 ) } ;
                    } )

      soMenu:addItem( { "~Seek" ,   ;
                        {|mp1,mp2,obj| SeekCustomer( obj:cargo ) } ;
                    } )

      soMenu:addItem( { "~Delete" , ;
                        {|mp1,mp2,obj| DeleteCustomer( obj:cargo ) } ;
                    } )

      soMenu:addItem( { "S~ave" , ;
                        {|mp1,mp2,obj| obj:cargo:writeData()  } ;
                    } )

      soMenu:addItem( MENUITEM_SEPARATOR )

      soMenu:addItem( { "~First" , ;
                        {|mp1,mp2,obj| DbGoTop() } ;
                    } )

      soMenu:addItem( { "~Last" , ;
                        {|mp1,mp2,obj| DbGoBottom() } ;
                    } )
```

```
            soMenu:addItem( MENUITEM_SEPARATOR )

            soMenu:addItem( { "~Previous" , ;
                              {|mp1,mp2,obj| DbSkip(-1) } ;
                            } )

            soMenu:addItem( { "~Next" , ;
                              {|mp1,mp2,obj| DbSkip(1) } ;
                            } )

            // menu items are disabled after Bof() or GoTop()
            soMenu:disableTop    := { 6, 9 }

            // menu items are disabled after GoBottom()
            soMenu:disableBottom := { 7, 10 }

            // menu items are disabled at Eof()
            soMenu:disableEof    := { 1, 2, 3 }

         ENDIF

      RETURN soMenu
```

A code block is defined for each menu item in the context menu. This code block is executed when the user selects the menu item. Many of the code blocks control database navigation using functions such as DbSkip(), DbGoTop(), and DbGoBottom(). The DataDialog object is automatically notified of these operations (its *:notify()* method is called) since it is registered in the work area. The context menu itself can only be activated on the DataDialog object (child window) which currently has focus. The menu is activated with a right mouse click that must occur within the DataDialog window. The DataDialog window activates its context menu through the following callback code block (see MDICUST.PRG file, function Customer()):

```
drawingArea:RbDown := {|mp1,mp2,obj| ;
     ContextMenu():cargo := obj:setParent(), ;
     ContextMenu():popup( obj, mp1 ) }
```

The *:drawingArea* is the drawing area of the DataDialog window. The ContextMenu() function is shown above. This function returns the contents of the STATIC variable *soMenu*, which is the context menu. The code block parameter *obj* contains a reference to the Xbase Part that is processing the event xbeM_RbDown (right mouse button is pressed). In this case, this is the drawing area of the DataDialog (*:drawingArea*) and the expression *obj:setParent()* returns the DataDialog object that is assigned to the *:cargo* instance variable of the context menu. This all occurs before the context menu is displayed using the method *:popUp()*. The current mouse coordinates (relative to *obj*) are contained in *mp1*. This allows the return value of ContextMenu() (the context menu) to be displayed at the position of the mouse pointer.

When a menu item is selected in the context menu, the DataDialog object where the context menu is activated is always contained in the *:cargo* instance variable. This DataDialog object has the focus (otherwise it would not react to a right mouse button click). The DataDialog object with the focus was previously selected via the callback code block *:setDisplayFocus* which sets the appropriate work area as the current work area. Database navigation can occur in the context menu by simply calling DbSkip() or DbGobottom() without the work area where the movement is to occur being specified. The work area is selected by the DataDialog object when it receives the focus. The context menu can only be activated on a DataDialog object that has the focus because only the DataDialog object with focus reacts to the event xbeM_RbDown and the context menu is only activated in the callback code block *:RbDown*.

This discussion outlines program control via a context menu as it is used in the example application MDIDEMO (it may be easier to follow by stepping through the code in the debugger). In conclusion, a context menu can be an important control element in a GUI application. Generally a context menu is not specific to a work area but calls functionality that must operate regardless of the work area. In short: a context menu controls an Xbase Part.

15. The Xbase++ Graphics Engine (GRA)

This chapter describes the Xbase++ GraphicsEngine (GRA engine) which provides the programmer a convenient environment for graphic output. The GRA engine provides functional access to the graphic output system. It includes a total of 32 functions which can be used to easily create simple, up-to-date business graphics such as bar charts and pie diagrams. The GRA engine can also display bitmaps and metafiles and modify the appearance of dialog elements (Xbase-Parts).

15.1. Basics for graphic output

Using the GRA engine requires that an Xbase++ program be linked in for the GUI mode. Graphic output can not be done in VIO mode (see the chapter "Compiling and linking"). All graphic functions begin with the prefix "Gra". They often require graphic coordinates which are generally specified in "pixel units". A pixel is a dot on the screen. The default size of a window in which an Xbase++ application is run is 640x350 pixels. The origin of the graphic coordinate system (the point 0,0) is at the bottom left corner of a window.

In text mode (VIO mode), the coordinates of the screen cursor are determined using the functions Row() and Col(). The origin of the coordinate system in text mode is the upper left corner of the window and the default size of a window is 25 rows and 80 columns. When programming using graphic functions, a different graphic coordinate system must be used. It has a different origin than the text mode coordinate system, and row and column coordinates are not used. Graphic X and Y coordinates are used instead. The XY coordinates are in relation to the geometric X axis (horizontal) and Y axis (vertical).

Graphic output does not involve a screen cursor such as the one used in text mode to identify the current position on the screen. Instead, graphics mode incorporates a "pen". The pen position in graphics mode is similar to the cursor position in text mode. The pen position is defined as a point in the coordinate system and the cursor position is defined in terms of its row and column position. In graphics programming, a point is always represented by an array containing two elements which represent the X and Y coordinates for a point. The following code shows a comparison of cursor and pen position:

```
SetPos( 10, 20 )             // position cursor in text mode
? Row()                      // result: 10
? Col()                      // result: 20

GraPos( , {20,10} )          // position pen in graphics mode
? GraPos()                   // result: { 20, 10}
```

Note: The origin of the coordinate system for graphic output is lower left while in text mode it is upper left. The unit for the coordinate system is "pixel". One pixel (or point) is specified using a two-element array.

Graphic output generally occurs in the current window. When the output is not in a window (for example, it may be sent to a printer), the output mechanism for graphics must be taken into account. Graphic output occurs only in what is called a "presentation space" (Note: in Xbase^{++}, presentation spaces are provided by the XbpPresSpace() class). A presentation space contains the part of the definition for graphic output that is independent of the output device. A graphic can look different depending on the output device used for the display. Example: he display of a graphic on the screen occurs in a window. The unit for the coordinate system is the "pixel". When the graphic is output to a printer, the unit "0.1 cm" could be selected instead of pixel. The output of the graphic on the printer would then differ considerably from the output in the window.

In the discussion of graphic output, the device independent output must be distinguished from device dependent output. A presentation space defines everything required for device independent display of graphics. This includes, for example:

- Origin and unit of the coordinate system
- Colors for the display
- Type of line for lines
- Fill pattern for areas
- Font type and size for characters
- Size of the display area

The presentation space can be thought of as an abstract drawing area for graphic output (a presentation space can also be thought of as similar to an empty piece of paper that can be drawn on). It defines all attributes for graphic output that are independent of the output device. An output device is, for example, a window (the screen) or a printer. A physical output device is called a "device context". A device context contains all device dependent attributes. To display graphic output, a presentation space (device independent) must be combined with a device context (device dependent). Nothing drawn in the presentation space is visible. Only by linking the presentation space with a device context can graphic output in the presentation space be made visible on the physical output device.

As long as graphic output occurs only in an XbpCrt window, the presentation space does not need to be dealt with because that window has a presentation space associated with it and makes the device context available. In the description of the GRA engine and its functions other windows or output devices are not considered. They are described in later chapters about presentation spaces and graphic output devices. In all graphic functions the first parameter specifies the presentation space for drawing. The current XbpCrt window is the default presentation space so that neither a device context nor a presentation space need to be specified when drawing in the current window. If a window is created in the default size, a drawing area of 640x400 pixels is available. The pen position is at the point {0,0} after the window is created. Drawing output always begins at the current pen position. Because it sets the pen position, the function GraPos() is an important, fundamental function of the GRA engine. Another basic function is GraError() which determines whether graphic output was successful or an error occurred.

Basic functions for the GRA engine

Function	Description
GraPos()	Return or set pen position
GraError()	Return numeric error code for last graphic output

The GRA engine is dependent on the hardware and on installed device drivers in the same way as the operating system does. If graphic output does not occur as expected, GraError() tests whether the graphic operation is unsupported by the hardware or device driver.

15.2. Graphic primitives

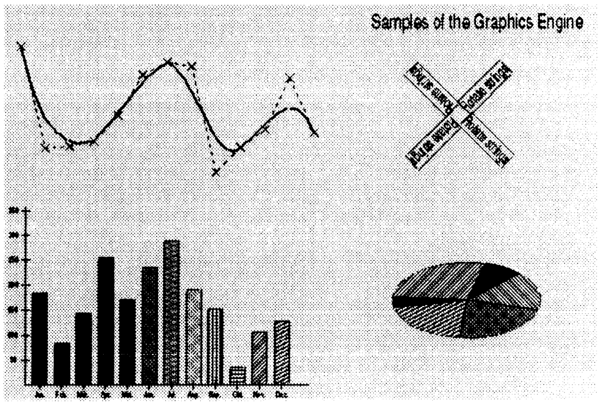
"Graphic primitives" are required to produce graphics. They are functions that draw elementary graphic figures, like lines, circles and rectangles. They always draw in a presentation space which displays itself in a device context that makes the result of graphic primitives visible on a physical output device.

Graphic primitives are the basic building blocks for graphic output. The GRA engine makes use of the six graphic primitives listed in the following table:

Graphic primitives

Function	Description
GraMarker()	Draws marker
GraLine()	Draws line
GraBox()	Draws box or rectangle
GraStringAt()	Draws character string
GraSpline()	Draws curve (Splines)
GraArc()	Draws circle, arc or ellipse
GraQueryTextBox()	Returns the coordinates of a character string

The functions in this table draw six elementary graphic figures: markers, lines, rectangles, circles, splines and character strings. The function GraQueryTextBox() is not a graphic primitive and does not display anything. This function determines the coordinates of the area occupied by a character string when it is output. This is significant for correct positioning of character strings on the display, especially with proportional fonts. The following illustration shows the result of various graphic primitives.



Output of graphic primitives

This illustration shows a few of the possibilities provided by the GRA engine through its graphic primitives. The illustration is created using Xbase^{++} and contains four examples. In the upper left quadrant lines, splines and markers are shown (the GraMarker(), GraLine() and GraSpline() functions). The upper right quadrant is drawn using GraStringAt(), GraQueryTextBox() and GraBox(). In the bottom half of the illustration two business graphs are shown. The bar graph is drawn mainly using GraBox() and GraLine() and the pie chart uses the function GraArc().

15.3. Attributes for graphic primitives

Attributes are set before drawing in order to modify the output of graphic primitives. These settings are valid until they are reset. For example, colors, line types and fill patterns for areas can be defined. The number of attributes differs for different graphic primitives. Attributes are defined in the form of an attribute array whose size and elements are determined using constants defined in the GRA.CH file. Also, the values that are entered in the elements of the attribute array are represented using #define constants.

The basic procedure for changing attributes of graphic primitives is shown in the following code:

```
aAttribute := Array( GRA_AL_COUNT )          // create empty attribute
                                             // array for lines

aAttribute[ GRA_AL_TYPE ]  := GRA_LINETYPE_SHORTDASH
aAttribute[ GRA_AL_COLOR ] := GRA_CLR_RED // select line type "short
                                             // dash" and the color red

GraSetAttrLine( , aAttribute )               // set line attributes
```

This code illustrates the naming convention used by the GRA engine for the #define constants. All the constants begin with the prefix GRA_. Generally another prefix is also included to identify the group of GRA_ constants and a suffix is included at the end which uniquely defines the constant. Attributes can be set for markers, lines, areas and character strings. Thus, there are four possible attribute arrays that are created using the four constants:

```
aAttrMarker := Array( GRA_AM_COUNT ) // attribute array for markers
aAttrLine   := Array( GRA_AL_COUNT ) // attribute array for lines
aAttrArea   := Array( GRA_AA_COUNT ) // attribute array for areas
aAttrString := Array( GRA_AS_COUNT ) // attribute array for string
```

The letters AM, AL, AA and AS are abbreviations for "attribute marker", "attribute line", "attribute area" and "attribute string". Each element of an attribute array defines a specific attribute. An element is identified by a #define constant which is prefixed by the attribute array prefix such as GRA_AM_, GRA_AL_, GRA_AA_ or GRA_AS_. An attribute is set by assigning a value to the appropriate element in the attribute array. For each attribute there is only a limited number of valid values which are again used in the form of #define constants.

```
aAttribute[ GRA_AL_TYPE ] := GRA_LINETYPE_SHORTDASH
```

In this line of code, the attribute "line type" receives the value for a short dashed line. Valid values for line types (more precisely, for the attribute GRA_AL_TYPE) are constants that are prefixed with GRA_LINETYPE_. For each attribute there is a group of #define constants like this that can be assigned as the value for the attribute.

Attributes are defined for a presentation space and therefore the attribute array must be passed to the presentation space after the attribute is inserted into the array. Only then are the attributes actually set and used for the graphic primitives during drawing. The functions used to assign the attribute arrays to the presentation space are listed in the following table.

Functions for attributes of graphic primitives

Function	Description
GraSetAttrMarker()	Sets attributes for markers
GraSetAttrLine()	Sets attributes for lines
GraSetAttrArea()	Sets attributes for areas
GraSetAttrString()	Sets attributes for character strings
GraSetColor()	Sets colors for all graphic primitives
GraSetFont()	Sets font for graphic output of characters

The GraSetAttr..() functions pass an attribute array to a presentation space (Note: these functions are coded in the GRASYS.PRG file. They call corresponding methods of an XbpPresSpace object). The attributes set for markers are used by the graphic primitive GraMarker(). The attributes for lines are used by GraLine() and GraSpline(). They are also used by GraArc() and GraBox() when a circle or a rectangle is drawn with a border (the border is a line). If a circle or a rectangle is to be filled with a pattern, the attributes for areas are used. These are defined by GraSetAttrArea(). The graphic primitives GraArc() and GraBox() use line attributes as well as area attributes. Attributes for character strings are used exclusively by GraStringAt(). The following tables give an overview of the attributes that can be set for markers, lines, areas and character strings.

#define constants for the attribute array for markers - GraSetAttrMarker()

Array element	#define I value	Description
GRA_AM_COLOR	GRA_CLR_*	Foreground color
GRA_AM_BACKCOLOR	GRA_CLR_*	Background color
GRA_AM_MIXMODE	GRA_FGMIX_*	Mix attribute for foreground color
GRA_AM_BGMIXMODE	GRA_BGMIX_*	Mix attribute for background color
GRA_AM_SYMBOL	GRA_MARKSYM_*	Marker symbol
GRA_AM_BOX	{nXsize, nYsize}	Size of the marker symbol

#define constants for the attribute array for lines - GraSetAttrLine()

Array element	#define group	Description
GRA_AL_COLOR	GRA_CLR_*	Foreground color
GRA_AL_MIXMODE	GRA_FGMIX_*	Mix attribute for foreground color
GRA_AL_WIDTH	GRA_LINEWIDTH_*	Line width
GRA_AL_TYPE	GRA_LINETYPE_*	Line type

#define constants for the attribute array for areas - GraSetAttrArea()

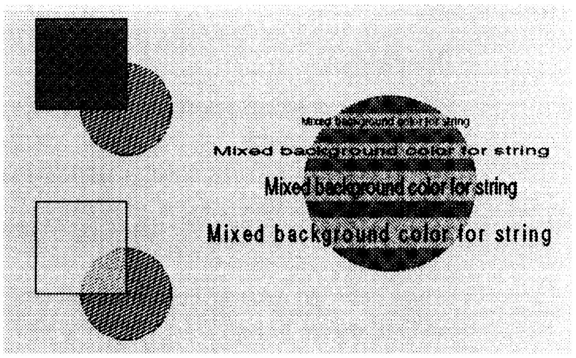
Array element	#define I value	Description
GRA_AA_COLOR	GRA_CLR_*	Foreground color
GRA_AA_BACKCOLOR	GRA_CLR_*	Background color
GRA_AA_MIXMODE	GRA_FGMIX_*	Mix attribute for foreground color
GRA_AA_BGMIXMODE	GRA_BGMIX_*	Mix attribute for background color
GRA_AA_SYMBOL	GRA_SYM_*	Fill pattern for areas

#define constants for the attribute arrays for character strings

Array element	#define I value	Description
GRA_AS_COLOR	GRA_CLR_*	Foreground color
GRA_AS_BACKCOLOR	GRA_CLR_*	Background color
GRA_AS_MIXMODE	GRA_FGMIX_*	Mix attribute for foreground color
GRA_AS_BGMIXMODE	GRA_BGMIX_*	Mix attribute for background color
GRA_AS_BOX	{nXsize,nYsize}	Size of each character
GRA_AS_ANGLE	{nX,nY}	Output angle of character strings
GRA_AS_SHEAR	{nX,nY}	Shear of characters (italics)
GRA_AS_DIRECTION	GRA_CHDIRN_*	Write direction
GRA_AS_HORIZALIGN	GRA_HALIGN_*	Horizontal alignment
GRA_AS_VERTALIGN	GRA_VALIGN_*	Vertical alignment
GRA_AS_EXTRA	nExtra	Distance between characters in character strings
GRA_AS_BREAK_EXTRA	nBreakExtra	Distance between words

Colors for graphic primitives

The function GraSetColor() defines the foreground and background color for the graphic primitives of the GRA engine. Colors defined using the function SetColor() are valid only for display in text mode and in hybrid mode. The GraSetAttr..() functions can individually set colors for markers, lines, areas and character strings that are then used instead of the globally defined colors. With the exception of lines, graphic primitives have a foreground and a background color. Lines have only a foreground color. The background color is not generally visible. It is used when displaying characters, which are always surrounded by a rectangle. The rectangle surrounding each character is drawn in the background color and the character itself (the letter) is displayed in the foreground color. The mix attribute for the background color must be defined in order to make the background color visible. Color mix attributes can be defined for the foreground color as well as for the background color. They determine how the colors of graphic primitives are mixed with each other when the output of a graphic primitive covers an already existing drawing.


Mixed background color for string
Mixed background color for string
Mixed background color for string
Mixed background color for string

Mix of foreground and background colors

In this illustration, the circles are drawn first. The mix attribute for the foreground color is changed from GRA_FGMIX_OVERPAINT (the default value, that causes the new primitive to cover the old) to GRA_FGMIX_OR prior to display of the lower square. When this mix attribute is set, the foreground color of the square is mixed with the foreground color of the existing drawing. When GRA_FGMIX_OR is set, the mix occurs using a bitwise OR between the color values of the existing colors and the colors of the new graphic being drawn. The area where the circle and square are combined is visible in the mix color and this section appears somewhat lighter than the rest of the circle. The color of the rest of the square in the illustration is the result of the color mix between dark gray (square) and pale gray (window color).

In the right part of the illustration, character strings are drawn over a circle. Within the circle, the background color for characters is visible. It causes the fill pattern of the circle to appear lighter. In this case, the mix color is a result of the value GRA_BGMIX_OR for the mix attribute of the background color of GraStringAt(). By default the mix attributes are set so that the foreground color paints over everything already displayed (GRA_FGMIX_OVERPAINT) and the background color remains unaffected (GRA_BGMIX_LEAVEALONE). The mix attribute for the foreground color is always significant when graphic primitives overlap each other and the shared section is to remain visible. When character strings are displayed, the mix attribute for the background color should also be considered, since both the foreground color (letter) and the background color (the rectangle which surrounds a letter) are shown.

A special value for the color mix attribute for foreground colors is GRA_FGMIX_XOR. The result of displaying graphic primitives when this value is set is that the graphic primitives are deleted from the screen if they are drawn at the same location a second time. Also, the resulting mix color of a graphic primitive which paints over an existing drawing depends on the color of the existing drawing and the color of the graphic primitive. The following two tables list the constants that can be used for the mix attribute of the foreground color and the background color:

Mix attributes for foreground colors

Constant	Description
GRA_FGMIX_OVERPAINT	Paints over existing colors
GRA_FGMIX_LEAVEALONE	Uses existing colors
GRA_FGMIX_OR	Mixes new and existing colors using bitwise OR of the color bits
GRA_FGMIX_XOR	Mixes new and existing colors using bitwise XOR of the color bits
GRA_FGMIX_AND	Mixes new and existing colors using bitwise AND of the color bits
GRA_FGMIX_NOTMERGESRC	Inverts mix color of GRA_FGMIX_OR
GRA_FGMIX_NOTMASKSRC	Inverts mix color of GRA_FGMIX_AND
GRA_FGMIX_NOTXORSRC	Inverts mix color of GRA_FGMIX_XOR
GRA_FGMIX_INVERT	Inverts existing color
GRA_FGMIX_NOTCOPYSRC	Inverts new color
GRA_FGMIX_SUBTRACT	Inverts new color and mixes it with existing color using bitwise AND of the color bits
GRA_FGMIX_MERGENOTSRC	Inverts new color and mixes it with existing color using bitwise OR of the color bits
GRA_FGMIX_MASKSRCNOT	Inverts existing color and mixes it with new color using bitwise AND of the color bits
GRA_FGMIX_MERGESRCNOT	Inverts existing color and mixes it with new color using bitwise OR of the color bits
GRA_FGMIX_ZERO	Resulting color is black (all color bits are set to 0)
GRA_FGMIX_ONE	Resulting color is white (all color bits are set to 1)

Mix attributes for background colors

Constant	Description
GRA_BGMIX_OVERPAINT	Paints over existing colors
GRA_BGMIX_LEAVEALONE	Uses existing colors
GRA_BGMIX_OR	Mixes new and existing color using bitwise OR of the color bits
GRA_BGMIX_XOR	Mixes new and existing color using bitwise XOR of the color bits

Set font for characters

The function GraSetFont(), along with GraSetAttrString(), affects the display of characters and character strings. GraSetFont() passes to a presentation space an XbpFont object defining the font for graphic display of characters. By default, the font "System VIO" is used. To change the font, an XbpFont object must be created and passed to the presentation space. The size can be defined when an XbpFont object is created (the size is in points). When an XbpFont object is created using XbpFont():new(), a font is not yet available, but is only loaded into memory by the method *:create()* (refer to the section "Basics for Xbase Parts" and the discussion of their life cycle). Also, an XbpFont object can only load fonts that are available as system fonts. A distinction must be made between fonts that are device dependent (for example, fonts only available for a printer) and those that are device independent.

It is recommended that fonts be loaded using XbpFont():new() and then modifid using GraSetAttrString(). The attributes for character strings allow the creation of diverse font displays. This begins with the size of individual characters and includes the angle in which a character is displayed relative to the horizontal axis.

Because of internal differences between XbpFont()and GraSetAttrString(), XbpFont() should be used only to load a font and GraSetAttrString() should be used to define the attributes for displaying characters. It should be noted that many attributes are valid only for vector fonts. A vector font is drawn as the outline of the letters using graphic primitives and then the outline is filled. A letter in a bitmap font consists of a raster image. Because of this, the characters of a bitmap font can not be automatically converted to italics. In order to display the characters of a bitmap font in italics, a separate italic bitmap font is required.

15.4. Graphic segments

A graphic segment contains one or more graphic primitives and allows the result of several graphic primitives to be redrawn without having to call the graphic primitives again. Graphic segments can also have a numeric ID and can be found based on a specified point in the coordinate system. For example, this can allow an application to determine which graphic segment a mouse click occurred in. The definition of a graphic segment is initiated using the function GraSegOpen() which returns the numeric ID for the new segment. All subsequent calls to graphic primitives are recorded in the segment until the definition of the graphic segment is completed using GraSegClose().

After the graphic segment is defined, all graphic primitives contained in the segment are redrawn when the function GraSegDraw() is called. The functions used to program graphic segments are listed in the following table:

Functions for graphic segments

Function	Description
GraSegOpen()	Initiates definition of graphic segment
GraSegClose()	Terminates definition of graphic segment
GraSegPriority()	Sets drawing order of graphic segment
GraSegDestroy()	Releases graphic segment
GraSegDraw()	Draws graphic segment
GraSegDrawMode()	Specifies drawing mode for graphic segments
GraSegFind()	Locates graphic segment based on position
GraSegPickResolution()	Sets resolution used in searching for segments

A graphic segment can be viewed as a complex primitive consisting of many graphic primitives that are executed between GraSegOpen() and GraSegClose(). Drawing, or making a graphic segment visible, depends on the drawing mode defined in the presentation space when the segment is opened with GraSegOpen(). There are three different modes that are set using the function GraSegDrawMode():

Drawing modes for graphic segments

Constant	Description
GRA_DM_DRAW	Primitives between GraSegOpen() and GraSegClose() are drawn but not stored
GRA_DM_RETAIN	Primitives between GraSegOpen() and GraSegClose() are stored but not drawn
GRA_DM_DRAWANDRETAIN	Primitives between GraSegOpen() and GraSegClose() are drawn and stored

The GRA_DM_DRAW mode draws but does not store the primitives defined in the segment. The segment can not be redrawn using GraSegDraw(). This mode is only useful when a drawing is to be stored in a metafile (see the sub-section "The metafile - XbpMetaFile()" in the section "Graphic output devices").

The GRA_DM_RETAIN drawing mode stores all graphic primitives in a segment, but does not display them. This allows complex drawings to be defined invisibly to the user step by step, and then made visible all at once using GraSegDraw(). In the third drawing mode, output occurs while the segment is being defined and all graphic primitives are already visible when the segment is closed using GraSegClose(). In this mode, the function GraSegDraw() only has to be called if the segment needs to be redrawn.

If graphic segments are defined for a window, only graphic primitives can be stored in the segment and not the attributes for the graphic primitives. When a segment is drawn, the attributes for points, lines, areas and characters that are set when GraSegDraw() is called are used for the display (Note: when graphic segments are stored in a metafile, the attributes are also stored).

The defined graphic segments are arranged in priority which increases in the order the segments were defined. The first segment defined has the lowest priority and the last segment defined has the highest priority. The priority determines the order in which graphic segments are drawn. If the function GraSegDraw() is called without parameters, all graphic segments are redrawn. The segment with the lowest priority is drawn first and the segment with the highest priority is drawn last. This is significant when graphic segments overlap each other during drawing. The segments with higher priority paint over segments with lower priority. The result is that segments with high priority appear "in front" and segments with low priority appear "behind" the segments with high priority. The priority can be thought of as the z axis in a three dimensional coordinate system. The higher the priority, the more toward the front the graphic segments appear if they overlap other segments. The function GraSegPriority() raises or lowers the priority of an individual segment in relation to a second segment.

A unique characteristic of graphic segments is the fact that they can be "found" based on their position. This is done using the function GraSegFind() which is passed the coordinates for a point as a parameter and returns an array containing the numeric IDs of all the graphic segments which encompass this point. The graphic segments can then be used for user interaction. After a mouse click, for example, the coordinates of the mouse pointer can be passed to the function GraSegFind() in order to identify all graphic segments that include the point where the mouse was clicked. The function GraSegFind() does not consider the exact coordinates of the mouse pointer (the point at the tip of the mouse pointer), but works with a virtual rectangle which is moved along with the mouse pointer. It is possible to find a graphic segment where the mouse pointer is near the segment but remains just outside. In this case the virtual rectangle around the mouse pointer overlaps the edge of the segment. The size of the virtual rectangle can be set using the function GraSegPickResolution() and determines the preciseness or resolution used by the function GraSegFind() in searching for graphic segments.

15.5. Graphic paths

A graphic path is very similar to a graphic segment. It is initiated using the function GraPathBegin() and terminated using GraPathEnd(). Graphic primitives which are called between these two functions describe the graphic path. The graphic path is defined in a manner similar to the definition of a graphic segment but it has an entirely different purpose: it defines the borders of an area.

After a graphic path is defined, the area described by it can be made visible by any one of three path operations. An overview of path operations is presented in the next table:

Functions for graphic paths

Function	Description
GraPathBegin()	Initiates definition of graphic path
GraPathEnd()	Terminates definition of graphic path
GraPathFill()	Fills graphic path
GraPathOutline()	Outlines graphic path
GraPathClip()	Specifies graphic path as a clipping path

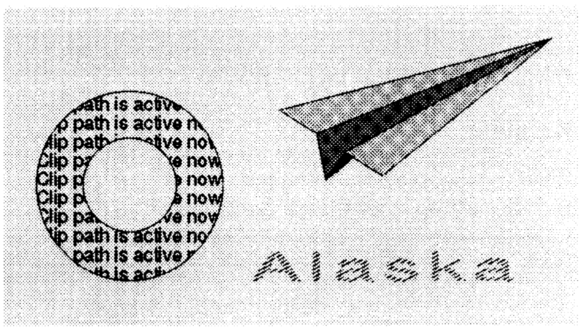
Graphic primitives that are called between GraPathBegin() and GraPathEnd() define the outline of an area that forms the graphic path. The primitives are not drawn and remain hidden. The defined area can then be filled with a color and/or pattern using GraPathFill(). The attributes for areas set with GraSetAttrArea() are used. The border of the defined area can be drawn using GraPathOutline() which makes the primitives visible. In this case, the line attributes set with GraSetAttrLine() are used.

After one of the two path operations (GraPathFill() or GraPathOutline()) are executed, the graphic path is discarded and is no longer available. When an area is to be outlined as well as filled, a graphic path can be defined within a graphic segment. The path can then be redefined using GraSegDraw(). Example:

```
nSegmentID := GraSegOpen()                  // open segment

   GraPathBegin()                           // initiate path
   /* execute graphic primitives */
   GraPathEnd()                             // end path

GraSegClose()                               // close segment

GraPathFill()                               // fill area

GraSegDraw( , nSegmentID )                  // redefine path

GraPathOutline()                            // outline path
```

Unlike GraPathOutline(), the path operation GraPathFill() requires that the area described by the path be closed. If this is not the case, the entire window is filled by GraPathFill().

A special path operation is performed using GraPathClip(). Anything drawn in the window is limited to the area defined by the graphic path. Output outside this area is not possible while the clipping path is set. This includes all graphic functions as well as the functions available in hybrid mode, like QOut() and Alert(). The clipping path is activated using GraPathClip(, .T.) and must be deactivated using GraPathClip(, .F.).



Graphic paths

This illustration shows some of the ways in which graphic paths can be used. On the left a clip path is defined by two concentric circles (using the function GraArc()). Within the clip path, a character string is drawn several times using GraStringAt(). The "paper airplane" consists of only seven points. The points are connected to each other using GraLine() and the graphic paths are defined and then outlined and filled with various patterns. Three fill patterns are used to illustrate the separate graphic paths, (the two "wings" are one graphic path). The text "Alaska" is drawn between GraPathBegin() and GraPathEnd() and then filled with a pattern using GraPathFill(). This shows that a graphic path itself can consist of several individually enclosed areas. Each letter forms a separate closed area.

15.6. Graphic transformations and raster operations

The GRA engine allows drawings created with graphic primitives to be transformed. "Transformed" means that a drawing can be turned, inverted, enlarged or reduced in size. There is an important distinction here between vector and raster images. A vector image is a drawing defined only by points in the coordinate system and how these points are connected with each other by lines. The lines can define the border for an area and the area can be filled with a color or a pattern. A raster image, however, is just an area filled with dots of color. The area is divided up by a raster and each dot in the raster has its own color. A raster image is generally called a "bitmap". A dot in a bitmap is called a "pixel".

There are three functions in the GRA engine for transforming vector images and a single function for performing raster operations.

These four functions are listed in the following table:

Functions for transformations and raster operations

Function	Description
GraRotate()	Calculates rotation transformation matrix
GraScale()	Calculates scaling transformation matrix
GraTranslate()	Calculates translation transformation matrix
GraBitBlt()	Performs operations with raster images (bitmaps)

Transformation from vector images

The six graphic primitives of the GRA engine are used to create vector images (except that when a bitmap font is set, character strings are drawn as raster images). A vector image can be transformed in any manner. There is the practical limitation that transformation of graphic primitives is not possible, but only the transformation of graphic segments (these contain graphic primitives). Thus, when drawings are to be transformed they must be defined within graphic segments.

The three possible graphical transformations of a vector image involve the operations of "rotating" (turn an image), "scaling" (enlarge or reduce an image) and "translating" (move, copy or invert an image). Each of these operations requires that a transformation matrix be calculated. The calculations for the three possible transformations are done using the functions GraRotate(), GraScale() and GraTranslate(). The three functions only calculate a transformation matrix for one of these three possible graphical transformations, they do not display anything. The transformation matrix itself is a two dimensional array with three rows and three columns. The array must be created before the transformation using GraInitMatrix(). GraInitMatrix() is a pseudo function defined in GRA.CH. It is translated into a two dimensional array which must be used for calculating a transformation matrix.

To make the transformation visible, the graphic segment must be redrawn. This occurs using the function GraSegDraw() and passing the transformation matrix containing the result of the calculations for the desired transformation to this function. Here is an example:

```
#include "Gra.ch"

PROCEDURE Main
   LOCAL  nSegment, i, aMatrix := GraInitMatrix()

   SetColor("N/W")                    // fill window with pale gray
   CLS

   nSegment := GraSegOpen()           // define segment
      GraBox( NIL, {200,150}, {300,230}, GRA_OUTLINE )
   GraSegClose()                      // create a box in a segment
   GraSegDraw( NIL, nSegment )        // display box

   FOR i:=1 TO 12                     // draw box 12 times rotating
                                      // each one 30 degrees left
      GraRotate ( NIL, aMatrix, -30, {200,150}, GRA_TRANSFORM_ADD )
      GraSegDraw( NIL, nSegment, aMatrix )
   NEXT
RETURN
```

In this example, a graphic segment is created that contains only one graphic primitive (GraBox()). Within the FOR..NEXT loop, the segment is drawn 12 times and each time a rotation of -30 degrees is calculated based on a turning point at the lower left corner of the box (the point {200,150}). The negative angle -30 degrees causes a rotation to the left (a positive angle would cause a rotation to the right). The example shows that it is essential to first calculate the transformation matrix. To calculate the transformation matrix, a matrix created by GraInitMatrix() is needed. The result of the calculation of GraRotate() is visible only after GraSegDraw() is called. The transformation matrix must be passed to the function GraSegDraw().

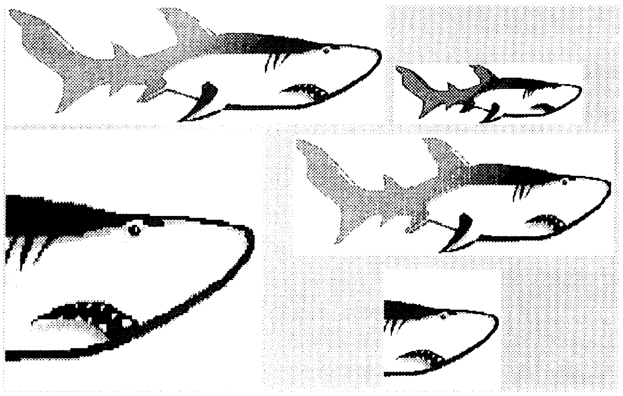
A graphic transformation requires a total of four steps. First, the transformation matrix must be created using GraInitMatrix(). The second step is defining the graphic segment containing the drawing to be transformed. Third, the transformation is calculated using GraRotate(), GraScale() or GraTranslate() to create the transformation matrix. Finally, the graphic transformation is displayed using GraSegDraw(). The transformation matrix previously calculated using GraRotate(), GraScale() or GraTranslate() must be passed to the function GraSegDraw().

Note: The two #define constants GRA_TRANSFORM_ADD and GRA_TRANSFORM_REPLACE affect the calculation of repeated transformations. If GRA_TRANSFORM_ADD is specified the transformation matrix is left unchanged after it is used to display the segment. GRA_TRANSFORM_REPLACE (default) causes the transformation matrix to be reset to the initial value of GraInitMatrix() when the display using the matrix calculation is complete. Using GRA_TRANSFORM_ADD, several transformations can effectively be added together. GRA_TRANSFORM_REPLACE begins each graphic transformation with the initial matrix predetermined by GraInitMatrix().

Important: These transformations are valid only for vector images. They can not be applied to raster images (bitmaps).

Transformation of bitmaps - Raster operations

Raster images (bitmaps) can be transformed in a manner similar to vector images, with the limitation that images in the form of a bitmap can not be turned (rotation is not possible with bitmaps). Bitmaps can be copied, enlarged or reduced in their entirety or in sections. The function GraBitBlt() handles these tasks. No transformation matrix needs to be calculated for transformations of raster images. The source and the target coordinates are passed to the function GraBitBlt() in the form of an array. GraBitBlt() processes the coordinates for the area of the bitmap which is to be copied and the coordinates for the area into which it is to be copied. The source coordinates correspond to the presentation space in which the raster image is already displayed and the target coordinates correspond to the presentation space into which it is copied. Both presentation spaces can be associated with the same device context. In this case, a raster image from the area of the source coordinates is copied to the area of the target coordinates. When the source and target coordinates describe areas of different sizes, automatic scaling of the raster image occurs. The following illustration shows the result of various raster operations using GraBitBlt(). The initial image is displayed upper left:



Raster operations with GraBitBlt()

To create this illustration, a bitmap is initially displayed in the upper left of the image and then the function GraBitBlt() is called four times using different source and target coordinates. The raster image is copied, enlarged, and reduced as a whole and in sections.

A special characteristic of the function GraBitBlt() is that the source and target areas of the raster operation can occur in two different presentation spaces. The output of the raster image depends on the device context linked to the presentation space when the display occurs. For example, it is possible to copy a raster image from a presentation space associated with a window to a presentation space associated with a printer device context. In this way raster images visible in a window can be output to a printer.

16. Presentation Spaces for Graphic Output

A presentation space is the central element for graphic output. Xbase^{++} uses the class XbpPresSpace() to provide a simple mechanism to access presentation spaces on the Xbase^{++} language level. The essential elements of a presentation space have already been described in the section "Basics for graphic output". As discussed, a presentation space makes an abstract drawing area available that contains all the device-independent information about a drawing. The output of a drawing on the physical output device depends on the device context associated with the presentation space. A device context manages an output device and is accessed in Xbase^{++} using instances of classes, like XbpPrinter() or XbpFileDev(). The interaction between the presentation space and various device contexts is described in the next section. This chapter discusses different presentation spaces. It begins with a description of the features common to all presentation spaces.

16.1. Coordinate system and view port

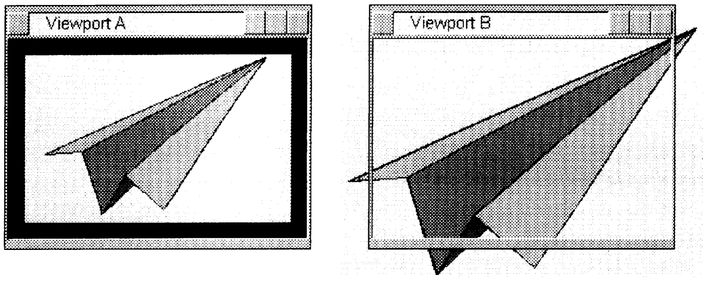
This section covers the most important aspect of a presentation space: the coordinate system. The presentation space determines the unit of measure for the coordinate system used to output graphics. The unit can be pixel, centimeter, inch or an arbitrary unit.

The presentation space determines the origin of the coordinate system (the point {0,0}) along with the unit of measurement for the output. All graphic output is in relation to the coordinate system of the presentation space and this in return relates to the coordinate system of the output device. When graphic elements are output, two coordinate systems are to be considered: the coordinate systems of the presentation space and the coordinate system of the device context.

The coordinate system of the presentation space is predetermined by the page size. The page determines the available space where graphic output can occur. Since all graphic output occurs in the presentation space, the output uses the page coordinates of the presentation space.

A presentation space displays the page in the device context. It uses what is called a "viewport". The entire page of a presentation space is completely displayed in the viewport. The viewport relates to the coordinates of the output device, meaning the size of the viewport determines what is visible on the output device. The size of the page, however, determines what can be displayed in the presentation space.

The page size of a presentation space as well as the size of the viewport can be set for a presentation space. The size of the page is determined using the method *:setPageSize()* and the size of the viewport is set using the method *:setViewPort()*. The page determines the display in the presentation space and the viewport determines the display on the output device (in the device context). The following illustration shows the effect of different size viewports with two pages of equal size and equal size output devices in two windows:



Viewport A is smaller than the window (viewport B > window)

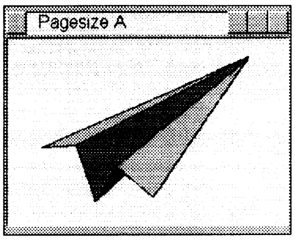
The illustration shows two equal size windows with a drawing area of approximately 260 * 200 pixels (excluding the title bar) which both display the same image. The only difference between the windows is the size of the viewport of their respective presentation spaces. The gray area in the illustration represents the viewport. In the left window the viewport is smaller than the window. In the right window the image spills out over the window border, because the viewport is larger than the window. Everything that is outside the window border of the right window would not be visible on the screen. The following program code shows how the viewport can be influenced:

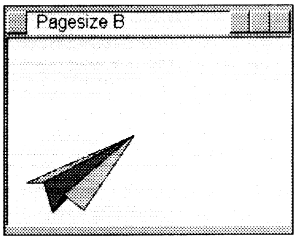
```
oPS1 := oWindowA:presSpace()
oPS1:setViewPort( {20, 20, 240, 180} )   // viewport < window

DisplayGraphic( oPS1 )                   // draw image

oPS2 := oWindowB:presSpace()
oPS2:setViewPort( {-20, -20, 280, 240} ) // viewport > window

DisplayGraphic( oPS2 )
```

In both windows, the image is drawn with the same coordinates. However, in the left window the viewport of the presentation space is smaller than the output device (the window). Correspondingly, the image is reduced. In the right window, the viewport is larger than the output device. The display in the window is enlarged and parts of the image are cut off. The gray area corresponds to the viewport. This program code illustrates that the coordinates of the viewport are specified as coordinates of the window. Thus, the viewport in the left window begins with the point {20,20}, which lies within the window. The viewport in the right window, however, lies outside the window because it begins at the point {-20,-20}.

A presentation space transfers graphic output through the viewport into the device context. The viewport of a presentation space always relates to the coordinate system of the output device. In the example, the output device is 260*200 pixels large and two different size viewports are defined. In this way images can be scaled in any way in the display, because the viewport determines the coordinates in the output device where the graphic output occurs. If the viewport does not have the same size as the output device, a graphic transformation is automatically performed during the output of the image to the device context.

Along with the viewport, the size of the abstract drawing area which represents the presentation space can be set for a presentation space. This occurs using the method *:setPageSize()* to define the size of the "page" in a presentation space. The page size affects the dimensions of the coordinate system in a presentation space.





Page of the presentation space in the right window is 4x larger

The illustration again shows two equal size windows with a drawing area of 260*200 pixels, and in both windows the same image is displayed. Here the page sizes of the two presentation spaces are different:

```
oPS1 := oWindowA:presSpace()
oPS1:setViewPort( {0, 0, 260, 200} )      // viewport = window
oPS1:setPageSize( {260, 200} )            // page size = window

DisplayGraphic( oPS1 )                    // draw image

oPS2 := oWindowB:presSpace()
oPS2:setViewPort( {0, 0, 260, 200} )      // viewport = window
oPS2:setPageSize( {520, 400} )            // page size > window

DisplayGraphic( oPS2 )
```

In both cases, the viewport is the same size as the window. This means that everything drawn in the presentation space also appears in the window. But the page size for the presentation space in the right window is four times larger than in the left window. In the left window the coordinate system of the presentation space extends from 0 to 260 in the x direction (horizontal) and from 0 to 200 in the y direction (vertical), meaning the page is

260*200 pixels large. In the right window the x axis of the presentation space extends from 0 to 520 and the y axis from 0 to 400. Since the image in both cases is displayed at the same coordinates in the presentation space, it is four times smaller in the presentation space of the right window than in the left window.

The method *:setPageSize()* defines the size of the page or the dimensions of the coordinate system in a presentation space. The method *:setViewPort()*, on the other hand, specifies the device coordinates of the device context where output occurs. Both methods cause an automatic transformation when the page size of the presentation space does not match the size of the output device or when the viewport does not match the dimensions of the coordinate system of the device context.

The two methods *:setPageSize()* and *:setViewPort()* offer two ways to scale a graphic. Both methods can be used for output in a window. The method *:setViewPort()* can only be used for screen output in a window and is limited to graphic output containing vector images. No automatic scaling is performed in the display of raster images (bitmaps) or raster operations (the function GraBitBlt()). The display of bitmaps or bitmap fonts always occurs in the coordinates of the device context. Characters which are displayed with a bitmap font can not be scaled and bitmaps must be explicitly scaled using GraBitBlt().

16.2. The intelligent presentation space - XbpCrt:presSpace()

An XbpCrt window is created in the AppSys() procedure as the default application window. It provides the Xbase^{++} hybrid mode where text-oriented and graphic screen output can be combined. This is the easiest way to develop programs for GUI.

As a special feature, the XbpCrt window already has a presentation space and serves as device context for it. An XbpCrt window forms a unit with its presentation space and thus allows easiest possible usage of the GraphicsEngine. The *:presSpace()* method of an XbpCrt window returns its presentation space. This method exists only in the XbpCrt class. As long as the application window is an XbpCrt window, Gra..() functions may be called without passing a presentation space as the first parameter, since it defaults to *SetAppWindow():presSpace()*.

Besides of providing the default presentation space, an XbpCrt window has some inbuilt intelligence which encapsulates complexities of the GUI that a programmer normally has to deal with. The presentation space of an XbpCrt 'remembers' what is currently displayed in the window. It buffers graphic output. This intelligence becomes obvious when an XbpCrt window is covered temporarily by another window and then brought back into the foreground. In this case, the covered part of the XbpCrt window must be refreshed or repainted, respectively. To repaint itself, a window receives the xbeP_Paint event from the operating system. It is normally a programmer's task to implement code for a window's repaint. This is not the case with XbpCrt windows. It processes xbeP_Paint messages on its own and causes its presentation space to redisplay lost parts of the window's contents.

Due to its inbuilt intelligence, the presentation space of an XbpCrt window guarantees all visible output to be redisplayed when the XbpCrt window is covered temporarily and brought into the foreground again. Assume the following line of code:

```
GraLine( , {0,0}, {640,400} )
```

This call to the GraLine() function draws a line, diagonal from the lower left to the upper right corner. It remains visible even if other windows are clicked to the forground temporarily. The line is redisplayed when the XbpCrt window regains focus, and no code must be implemented to process the xbeP_Paint event.

16.3. The high speed presentation space - Xbp:lockPS()

For common graphic output with Gra..() functions, a so called Micro presentation space is used (Micro PS). The operating system itself has a buffer of Micro PSs and supplies them on request to an application program. A Micro PS is optimized for high speed graphic output and is reused by the operating system. When a window uses a Micro PS and graphic output is done, the Micro PS is returned to the operating system where it is collected in a buffer for further requests. Therefore, time-consuming allocation and release of memory does not occur with a Micro PS.

A Micro PS can be used by all Xbase Parts subclassed from the XbpWindow class. It is requested and returned by the *:lockPS()* method. An Xbase Part has exclusive access to this Micro PS until it releases it with *:unlockPS()* and returns it to the operating system. On the Xbase^{++} language level, a Micro PS is represented by an object. It has the same methods as the XbpPresSpace class, except of the *:create()*, *:configure()* and *:destroy()* methods. Due to technical details of the operating system, a Micro PS can only be created by *:lockPS()* and must be released with *:unlockPS()* (note: the XbpCrt class is not subclassed from XbpWindow. Therefore, a Micro PS cannot be used by an XbpCrt window). The general usage of a Micro PS follows this pattern:

```
oPS := oXbp:lockPS()            // request Micro PS
Gra???( oPS, ... )              // graphic output
oXbp:unlockPS( oPS )            // release Micro PS
```

A drawing is displayed in a Micro PS in the same way as in the presentation space of an XbpCrt window. However, due to its performance optimization, a Micro PS recognizes neither graphic segments nor graphic paths. Graphic output is limited to simple Gra..() functions. The complex functions GraSegOpen() and GraPathBegin() plus corresponding functions for managing graphic segments and paths cannot be used with a Micro PS. Furthermore, a Micro PS does no screen buffering. This requires an Xbase Part to react to the xbeP_Paint event. When using a Micro PS, code must be implemended for graphic output to be displayed after repaint events. This code can be programmed either in a function that is called by the *:paint* callback code block, or it must be programmed in the *:paint()* callback method of a derived class. An example of using a Micro PS is found in ...\SAMPLES\XPARTS\MICROPS.PRG.

16.4. The complete presentation space - XbpPresSpace()

Instances of the XbpPresSpace class are complete presentation spaces (PS). They can be used by all functions of the GRA Engine. This includes complex graphic output that consists of graphic segments and/or paths. If a complex graphic is to be displayed in Xbase Parts subclassed from XbpWindow, an XbpPresSpace object must be connected to the window device of this Xbase Part. If a drawing is displayed in an XbpDialog window, for example, the output device is *XbpDialog:drawingArea* and the PS must be connected to its window device:

```
oDialog := XbpDialog():new()              // Create an XbpDialog
oDialog:create( ,, {0,0}, {600,400} )     // window

oPS     := XbpPresSpace():new()           // Create a PS
oDevice := oDialog:drawingArea:winDevice() // Get the device context

oPS:create( oDevice )                     // Link device context
                                          // to PS
GraLine( oPS, {0,0}, {600,400} )          // Display drawing
```

The example demonstrates different steps necessary for using a complete PS. The device context (the output device) is returned by the *:winDevice()* method, and linked to the PS by passing it to the *:create()* method. Then a drawing can be created in the PS using Gra..() functions and the output appears in the XbpDialog window. Since the PS does not buffer screen output, the code that creates a drawing must repeatedly be executed after xbeP_Paint events.

17. Graphic output devices

In order to make the graphic output in a presentation space of a GUI application visible, the presentation space must be linked to a device context. A device context contains all device dependent information required for output of a drawing on an output device. In most cases the display of graphics occurs in a window on the screen. A window represents a device context. The GRA engine and its functions do not need to consider either a presentation space or a device context as long as the output occurs on the screen in an XbpCrt window. But the output of a graphic can also occur to a file or printer. In this case, a corresponding device context must be associated with a presentation space. Output to a printer is extremely device dependent. It requires the correct installation of a printer driver in the workplace shell. A printer is used for output in Xbase++ by an object of the XbpPrinter() class. XbpPrinter objects prepare a device context for graphic output to a printer. If the output is to be sent to a file, a decision must be made whether the file will contain a metafile or a raster image (bitmap). Both types of files contain graphic data and there are object classes to manage the file formats for both file formats. These classes are: XbpMetafile() and XbpBitmap(), respectively.

17.1. The printer - XbpPrinter()

Objects of the XbpPrinter class are used for printed output. This requires a printer driver appropriate for the physical printer be installed.

Basics for printing of graphics

The XbpPrinter class creates a connection to printer drivers, or printer objects, respectively, installed on the system. If a printer object is not correctly configured, correct graphic output to the printer from Xbase++ is not guaranteed. Printer objects are identified in Xbase++ by their name which is displayed underneath a printer icon. An XbpPrinter object controls a printer object and represents the device context for a presentation space in which graphic information is displayed. To set up for printing an image, therefore, the printer device context (XbpPrinter object) is associated with a presentation space. The following user-defined function is suitable for this:

```
FUNCTION PrinterPS( cPrinterObjectName )
   LOCAL oPS, oDC := XbpPrinter():new()

   oDC:create( cPrinterObjectName )

   oPS := XbpPresSpace():new()
   oPS:create( oDC, oDC:paperSize(), GRA_PU_LOMETRIC )
RETURN oPS
```

The user-defined function PrinterPS() creates a presentation space associated with a printer device context. It receives as a parameter the name of one of the printer objects installed on the system as a character string and creates an XbpPrinter object that maintains the connection to the printer object. In the function, a new presentation space is created, and the XbpPrinter object is provided as the device context. The return value of the method *:paperSize()* is used for the page size of the presentation space. This assures that the presentation space and the device context have the same size coordinate systems. Since the paper size is always returned by the method *:paperSize()* in the units of 1/10 millimeter, the constant GRA_PU_LOMETRIC is specified in the call to *oPS:create()* to set the same units in the presentation space.

The output of graphics on the printer can be performed using the function PrinterPS() as shown in the following example:

```
oPS := PrinterPS()                   // presentation space with
                                     // default printer object
oPS:device():startDoc()              // start print output
                                     // (open spooler)
GraBox( oPS, {0,0}, {1000,1000} )    // draw box

oPS:device():endDoc()                // end print output
                                     // (close spooler)
```

These four lines of code show the basic procedure for printing graphics. A presentation space associated with an XbpPrinter object (printer device context) must be available. In the example, the method *:device()* returns the XbpPrinter object and begins the print output using the call to *:startDoc()*. Generally, print jobs are spooled and *:startDoc()* opens the spooler. All graphic output which occurs in the presentations space is sent to the spooler. The end of the print job is signalled by the method *:endDoc()* which closes the spooler.

The function PrinterPS() shows how a presentation space for output on the printer is created, and also shows the basic difficulty of graphic output: the coordinate systems of the output devices "screen" and "printer" differ in things like the units of the coordinate system of the two devices. The following program code illustrates this:

```
GraBox( oPS, {0,0}, {1000,1000} )
```

This call to GraBox() draws a square with an edge length of 1000 units. How and where the square is displayed depends on the device context associated with the presentation space *oPS* and on the units for the coordinate system. The device unit for the screen is "pixel" and the square would only partially be visible on screen except at extremely high screen resolution. If a printer is the device context, the unit is 1/10 millimeter and the square would be displayed with 10cm edge length which fits the most common paper formats.

Print text as graphics

The problem of different coordinate systems and units becomes especially obvious when outputting text or characters. The function GraStringAt() draws a character string in a presentation space. The font which is set in the presentation space is significant. There are fonts which can only be output on the screen, others that can only be output on a printer, and others which can be output on either device. When text is output on a printer, fonts which are only for display on screen obviously can not be used. The following example program illustrates this problem:

```
#include "Gra.ch"
#include "Xbp.ch"

PROCEDURE Main
   LOCAL oWindowPS, oPrinterPS, oFont, aFontList
   LOCAL i, imax, nY, nPointSize, cText, aSize

   // presentation space for window and printer
   oWindowPS  := SetAppWindow():presSpace()
   oPrinterPS := PrinterPS()

   // create font object
   oFont := XbpFont():new( oWindowPS )

   // read list of available fonts
   aFontList := oFont:list()

   // output fonts on the printer
   oPrinterPS:device():startDoc()

   imax       := Len( aFontList )
   nY         := oPrinterPS:device():paperSize()[2] - 100
   nPointSize := 1

   FOR i:=1 TO imax

      // print only universally valid vector fonts
      IF aFontList[i]:generic .AND. aFontList[i]:vector

         oFont := aFontList[i]
         oFont:nominalPointSize := ++nPointSize

         // create text to print
         cText := Str( nPointSize, 2 ) + "." + ;
                       oFont:compoundName

         // set font for printer
         GraSetFont( oPrinterPS, oFont )
```

```
            // calculate size and new y position of the string
            aSize := GraQueryTextBox( oPrinterPS, cText )
            nY    := nY - aSize[1,2]

            // print string and release font
            GraStringAt( oPrinterPS, {20,nY}, cText )
            oFont:destroy()

         ENDIF
      NEXT

      oPrinterPS:device():endDoc()
   RETURN
```

This example program prints the size and name of all vector fonts that can be output on the printer. First, the presentation space of the application window is determined. The function PrinterPS() creates a presentation space which is linked with a printer device context. An XbpFont object is then generated to manage the fonts. The call *oFont:list()* returns an array containing the font objects for all fonts that can be displayed in a window. The print output then begins. The text to be printed is created in the FOR..NEXT loop. It is only printed when the instance variables *:generic* and *:vector* of a font object in the array *aFontList* both contain .T. (true). This is the prerequisite for fonts that can be displayed in a window and can also be output to scale on a printer. Within the FOR..NEXT loop, font objects which fulfill this prerequisite are passed using GraSetFont() to the presentation space *oPrinterPS* which is associated with the printer device context. At this point in the example program, the presentation space knows which font is to be printed and the metric data for the size of the letters. Since the point size for a font is increased by one after each pass through the loop, the y position of the character string must continually be adjusted to the height of the letters. The height of the character string *cText* is determined using the function GraQueryTextBox().

The example program touches on all the problems that must be considered in programming graphic output of text for a printer. Text is output in graphics mode using the function GraStringAt(). In order for the characters to be printed on paper, the output must occur in a presentation space linked with a printer device context (an XbpPrinter object). The font used for the print output is set using an XbpFont object. The available fonts are determined using the *:fontList()* method of a font object. In the example, all fonts which can be displayed in a presentation space for a window are tested, since the presentation space of the application window is provided to the XbpFont object (*XbpFont():new(oWindowPS)*). Only device independent vector fonts (*:generic* and *:vector* are .T.) can also be used for printing. When *:generic* and *:vector* are .T., this indicates that the font can be scaled correctly during printing. The exact position where a character string *cText* is printed in the example program must be calculated after each pass through the loop using the function GraQuerytextBox(), since the point size of the font is continually changed.

Using XbpPrinter objects to output text or graphics can be summarized as follows:

- The printer object must be installed and configured correctly.

- The output must occur in a presentation space which has an XbpPrinter object as its device context.

- The coordinate system of the XbpPrinter object is based on the unit of 1/10 millimeter.

- The font for the output of text is set using an XbpFont object.

- The size of a character string is determined by GraQueryTextBox().

These five points must always be considered when using XbpPrinter objects or sending graphic output to a printer.

17.2. The metafile - XbpMetaFile()

Metafiles play a central role in storing and exchanging graphic data and drawings. Metafiles store images in the same form they are drawn in the presentation space and allow the drawing to be redisplayed as necessary in the same or in another application. Images can be stored and printed in the format of the metafile, they can be exchanged between two work stations in this format, and they can be temporarily stored in the Clipboard while being transferred from one application to another.

The sequence of graphic primitives used to create a drawing is stored in the metafile. This includes the coordinates of the graphic primitives and their attributes. A metafile generally contains a vector image, because it stores a sequence of graphic primitives that can be "played back" again after the metafile is loaded in a presentation space. Metafiles are also used to exchange drawings between two work stations. As long as the metafile contains a drawing created using only graphic primitives, a drawing can be output on different screens or printers. Raster images can also be stored in a metafile (see function GraBitBlt()), and in this case the correct redisplay of a metafile on different output devices is not guaranteed since it is device dependent.

Xbase^{++} allows loading, displaying, storing and creating metafiles. For loading and displaying, objects of the class XbpMetaFile() are used. They manage metafiles that are already stored on a disk. The following lines of code show how metafiles are displayed in a window:

```
oMF := XbpMetaFile():new():create() // create metafile object
oMF:load( "METAFILE.MET" )          // load metafile

oPS := SetAppWindow():presSpace()   // get presentation space
                                    // from the window

oMF:draw( oPS )                     // redisplay the contents of the
                                    // metafile in presentation space
```

This procedure is designed to load existing metafiles into an Xbase++ application. But it is also possible to store graphic output created by an Xbase++ application in a metafile. To accomplish this, an XbpFileDev object must be created. It represents the device context where the graphic output of the Gra...() functions is recorded. Example:

```
PROCEDURE Main
   LOCAL oPS, oDC, oMF

   SetColor( "N/W" )                   // fill window pale gray
   CLS

   oDC := XbpFileDev():New():Create()  // create device context
                                       // create presentation space
   oPS := XbpPresSpace():New():Create( oDC ) // and combine with DC

   GraBox( oPS, {10,10}, {400,100} )   // graphic output
   GraStringAt( oPS, {20,50}, ;
                "Image will be stored in metafile")
   oPS:configure()                     // detach device context

   oMF := oDC:metaFile()               // create XbpMetaFile object
   oDC:destroy()                       // release device context

   IF File( "Test.met" )               // assure that no TEST.MET
      FErase( "Test.met" )             // file exists
   ENDIF

   oMF:save( "Test.met" )              // save image in file
   WAIT "Metafile is created, press key..."

   oMF:= XbpMetaFile():new():create()  // new MetaFile object
   oMF:load( "Test.met" )              // load file

   oPS:= SetAppWindow():presSpace()    // get PS from window
   oMF:draw( oPS )                     // output in this PS

   WAIT                                // wait for keypress
RETURN
```

In this example, an XbpFileDev object is created as a device context and associated with a presentation space. The graphic output is performed in this presentation space. When the output is completed, the device context is detached from the presentation space using a call to the *:configure()* method of the presentation space. The XbpFileDev object is now able to The method *:save()* saves this graphic information in a new metafile.

17.3. The raster image - XbpBitmap()

A raster image (bitmap) contains graphic information in the form of pixels. All the pixels in a bitmap that are required to display the image are saved. The XbpBitmap class provides the ability to create and display bitmaps. Raster images saved in a file are connected to an executable file as a resource and can be identified, loaded and displayed using an XbpBitmap object based on the numeric resource ID. Using XbpBitmap objects in this way starts with the declaration of a bitmap file (BMP file) as a resource within an RC file. For example:

```
/* Type of resource    Resource ID     File name */
   ICON                1         =     "\XPP\RESOURCE\XPPPMT.ICO"
   POINTER             2         =     "\XPP\RESOURCE\XPPPOINT.PTR"

   BITMAP              2000      =     "\XPP\BITMAP\PHOTO.BMP"
```

If this resource file is linked to the executable file by RC.EXE, an XbpBitmap object can load and display the resource 2000. This is done in the following example:

```
PROCEDURE Main
   LOCAL oBMP, oPS

   SetColor( "N/W" )
   CLS

   // get presentation space from XbpCrt window
   oPS := SetAppWindow():presSpace()

   // create XbpBitmap for this PS
   oBMP:= XbpBitmap():new():create( oPS )

   // load bitmap
   oBMP:load( NIL, 2000 )

   // display bitmap in the PS
   oBMP:draw( oPS, {50,100} )

   WAIT
RETURN
```

Before an XbpBitmap object can be created, the presentation space and its device context must exist that will be used for output of the raster image. In the example, output occurs in the presentation space of an XbpCrt window and this represents the output device. The presentation space must be specified in the call to the method *:create()* which prepares the XbpBitmap object for the display on the screen. The raster image from the PHOTO.BMP file with the resource ID 2000 is then loaded using the method *:load()* and displayed in the presentation space of the window using *:draw()*.

An XbpBitmap object is also used if a raster image is to be temporarily created in main memory. The methods *:presSpace()* and *:make()* are provided for this situation. An XbpBitmap object is associated with a presentation space using *:presSpace()*. Graphic output can then occur directly in the raster image, and is not visible on the screen. After an XbpBitmap object has received its own presentation space using the method *:presSpace()*, the size of the raster image must be specified using the method *:make()*. The memory for screen information (pixels) is then allocated. Graphic output can then occur in the presentation space of the bitmap.

The most important area for this use of XbpBitmap objects is in buffering graphic mode screen output. The entire display in a window or a section of a window can be stored as a raster image of an XbpBitmap object and redisplayed at a later time. Graphic information from the presentation space of a window is copied to the presentation space of an XbpBitmap object using the function GraBitBlt(). This is shown in the following example in the code for the user-defined function GraSaveScreen():

```
PROCEDURE Main
   LOCAL oPS, oBitmap

   // get presentation space from the XbpCrt window
   oPS := SetAppWindow():presSpace()

   // draw box
   GraBox( oPS ,{10,10}, {100,100} )

   // save box
   oBitmap := GraSaveScreen( oPS, {10,10}, {91,91} )

   // delete display in the window again
   CLS
   WAIT "Box is saved"

   GraRestScreen( oPS, {10,10}, oBitmap )
   WAIT "Box is redisplayed"
RETURN

FUNCTION GraSaveScreen( oSourcePS, aPos, aSize )
   LOCAL oBitmap   := XbpBitmap():new():create( oSourcePS )
   LOCAL oTargetPS := XbpPresSpace():new():create()
   LOCAL aSourceRect[4], aTargetRect

   aSourceRect[1] := aSourceRect[3] := aPos[1]
   aSourceRect[2] := aSourceRect[4] := aPos[2]
   aSourceRect[3] += aSize[1]
   aSourceRect[4] += aSize[2]

   aTargetRect    := {0, 0, aSize[1], aSize[2]}

   oBitmap:presSpace( oTargetPS )
   oBitmap:make( aSize[1], aSize[2] )

   GraBitBlt( oTargetPS, oSourcePS, aTargetRect, aSourceRect )
RETURN oBitmap
```

The function GraSaveScreen() receives the presentation space of a window in the parameter *oSourcePS*. This contains the graphic information to be saved by copying it into the presentation space *oTargetPS*. An XbpBitmap object is created as an output device which is suitable for output on the screen (*oSourcePS* is passed to *:create()*). For the XbpBitmap object to be able to save graphic output, it needs its own presentation space, which is passed to the object using the method *:presSpace()*. The newly created presentation space is referenced by *oTargetPS*. The size of the raster image is specified in *:make()*. The size corresponds to the parameter *aSize* which defines the dimensions of the screen section to be saved in the x direction and y direction. Finally, the function GraBitBlt() copies the pixels from the presentation space of the window into the presentation space of the XbpBitmap object.

The user-defined function GraRestScreen() is used to redisplay the saved section of the screen. The raster image managed by an XbpBitmap object is redisplayed in the presentation space of a window. Calling the method *:draw()* is sufficient to accomplish this task:

```
FUNCTION GraRestScreen( oTargetPS, aPos, oBitmap )
   oBitmap:draw( oTargetPS, aPos )
RETURN NIL
```

An XbpBitmap object needs a presentation space where the raster image managed by the object is output. When a bitmap is attached as a resource to an executable file, calling the method *:load()* is sufficient to allow display of the raster image. This method implicitly requests a presentation space for the XbpBitmap object. If, however, a raster image is to be created (during screen buffering, for example), the XbpBitmap object must have its own presentation space which is provided to the object by the method *:presSpace()*. This method must be executed before the call to *:make()*. *:make()* defines the dimensions for a raster image and the memory required to hold all the pixels of the raster image. The required memory can only be determined when a presentation space associated with a device context is provided.

17.4. Using windows as output devices

In the discussion of windows serving as output devices it is necessary to distinguish an application window (as provided by the XbpCrt and XbpDialog classes) from all other Xbase Parts having a visual representation. These, of course, are windows as well. The easiest way to achieve graphic output is by using an XbpCrt window. It serves as device context for its specialized presentation space and performs buffered screen output. Therefore, Gra..() functions can be used to draw in an XbpCrt window without the necessity of reacting to the xbeP_Paint event. This event is created by the operating system when a window needs to (partially) repaint itself.

Since an XbpDialog window does not buffer screen output, it must react to the xbeP_Paint message if Gra..() functions are used to draw in the window. The same is true for all other Xbase Parts. Regarding screen output, an XbpDialog window does not differ from a pushbutton or an XbpStatic object. The only difference is that an XbpDialog window is a compound object and drawing must be done in the drawing area below the title bar of the window (*XbpDialog:drawingArea*). In an XbpStatic element, for example, screen output appears in the object itself.

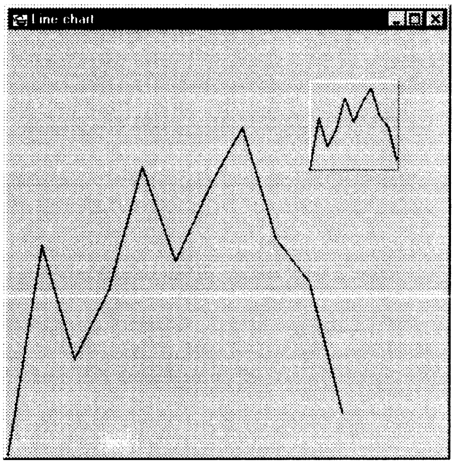
In order to enable an Xbase Parts to react to the xbeP_Paint event, a code block must be assigned to the *:paint* instance variable. This code block must call a routine where graphic output is implemented. As an alternative, a user-defined class can be derived from an Xbase Part. In this case, the *:paint()* method must be overloaded and code for graphic output must be implemented in this method. Either possibility requires a presentation space to be associated with the Xbase Part when Gra..() functions are called. This can be a Micro presentation space returned by the *:lockPS()* method, or an XpbPresSpace object which is linked to the device context of the Xbase Part. It is the return value of the *:winDevice()* method.

The different aspects of graphic output in windows, or Xbase Parts, are illustrated in the example program CHARTS.PRG which is located in the ..\SOURCE\SAMPLES\GRAFIC directory. This program draws a simple line chart by connecting a set of points with lines using GraLine():

```
// x and y values for a line chart in the range 0-300
aValues := ;
  { { 30, 184}, ;
    { 60,  84}, ;
    { 90, 144}, ;
    {120, 254}, ;
    {150, 170}, ;
    {180, 235}, ;
    {210, 289}, ;
    {240, 190}, ;
    {270, 152}, ;
```

```
     {300,  36}  }

LineChart( , aValues )

PROCEDURE LineChart( oPS, aPoints )
   GraPos( oPS, {0,0} )
   AEval( aPoints, {|aPos| GraLine( oPS, , aPos )} )
RETURN
```

This code would already work if an XbpCrt window was being used for display. The procedure LineChart() simply receives an array of values specifying xy coordinates for the points to be connected with lines. However, the CHART.PRG example program uses an XbpDialog and an XbpStatic object for display.



Output of the sample program CHART.PRG

The program draws the line chart first in the XbpDialog window and then in reduced size in the XbpStatic object (frame). The program code that displays the XbpDialog is given below. It consists of creation of the window, the definition of the *:paint* code block, display of the window and an event loop:

```
// Create window hidden
oDlg := XbpDialog():new(,,{10,10},{400,400},,.F.)
oDlg:title := "Line chart"
oDlg:create()
```

```
// Define :paint code block
oDlg:drawingArea:paint := ;
    {|mp1,mp2,obj| mp1 := obj:lockPS(), ;
              LineChart( mp1, aValues ), ;
                   obj:unLockPS( mp1 )  }

// Display window
oDlg:show()

// Process events
DO WHILE nEvent <> xbeP_Close
   nEvent := AppEvent( @mp1, @mp2, @oXbp )
   oXbp:handleEvent( nEvent, mp1, mp2 )
ENDDO
```

The *:paint* code block must be assigned to the drawing area of the XbpDialog object (*oDlg:drawingArea*) because it displays the drawing. The code block calls the LineChart() procedure which draws the chart. Evaluation of the code block, however, takes place in the *:handleEvent()* method when the xbeP_Paint event is returned from AppEvent(). The following steps lead to the display of the line chart:

```
Calling oDlg:show() creates an xbeP_Paint event in the event queue

AppEvent() returns xbeP_Paint and oXbp contains oDlg:drawingArea

oXbp:handleEvent( xbeP_Paint, mp1, mp2 ) evaluates the :paint code block

Eval( oXbp:paint, mp1, mp2, oXbp ) displays the line chart
```

The final result of the xbeP_Paint event is that *oDlg:drawingArea* is passed to the *:paint* code block as third parameter. Inside the code block, a Micro PS is requested using *:lockPS()* which gets passed together with *aValues* to the LineChart() procedure. Calling a drawing routine in this way via the *:paint* code block also guarantees that the procedure is executed again when the window is covered temporarily by another one and regains focus. In this case, the operating system creates a new xbeP_Paint event and the *:paint* code block is evaluated again.

Displaying the line chart in the XbpStatic object follows the same logic: LinChart() is called from the *:paint* code block. But instead of a Micro PS, an XbpPresSpace object is used which is linked to the window device context of the XbpStatic object:

```
// Create static frame inside oDlg:drawingArea
oXbp      := XbpStatic():new( oDlg:drawingArea,,{270,250}, {80,80} )
oXbp:type := XBPSTATIC_TYPE_RAISEDBOX
oXbp:create()

// Link presentation space to window device
oPS := XbpPresSpace():new()
oPS:create( oXbp:winDevice(), {310,310} )
```

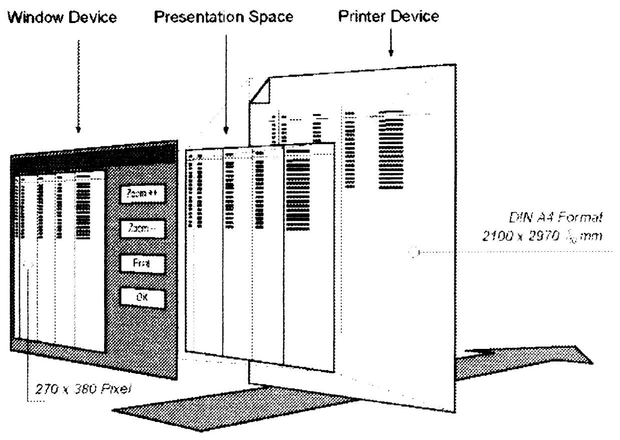
```
// Set viewport to the size of the frame
oPS:setViewPort( {0,0,80,80} )

oXbp:paint := {|| LineChart( oPS, aValues ) }
```

The line chart is drawn inside the XbpStatic object at the exact same coordinates as in the XbpDialog object. The reason why it is completely visible is given by page size and viewport of the presentation space. The page is large enough to display all points (310*310 pixel size vs. range of 0-300). The page is scaled to the viewport which is as large as the static frame (80*80 pixel). Therefore, the chart is reduced in size but completely visible.

17.5. Output to window and printer - WYSIWYG

A presentation space contains all device-independent information of a drawing. It can be linked to different output devices and therefore, a drawing can be displayed in a window or output to a printer. Depending on the device context a presentation space is linked to, printer output can be previewed in a window. In order to comply with the WYSIWYG rule (What You See Is What You Get), a drawing must be displayed in the window in correct scale. The proportions of a drawing displayed in a window must be the same as on paper after printing.

Displaying printer output in a window in correct scale involves three different levels of abstraction: one presentation space and device contexts for both window and printer. In addition, the device contexts for the two output devices have different units of measurement for their coordinate system. A window displays a drawing based on pixel while a printer uses units of 1/10th of a millimeter.



Previewing printer output in a window

This illustration is based on the example program PREVIEW.PRG. It demonstrates various aspects for correctly scaled output on different devices. The program lists database records in a window and can print them. The contents of database fields are drawn with the function GraStringAt().

On the left of the illustration, the application window is shown which is created by the example program. Database records are displayed in this window by an instance of the user-defined XbpPreview class, implemented in the example program. Its size is 270 x 380 pixel. On the right side of the XbpPreview object, there are pushbuttons that enlarge (zoom ++), reduce (zoom --) or print the display. If the display is enlarged, it can be scrolled by means of two scroll bars. The XbpPreview object provides the device context for screen output (window device).

A sheet of paper is shown on the right of the illustration. It represents printed output (the printer device) and has A4 format. This means its size is 2100 x 2970 units, one unit being 1/10th of a millimeter.

Between window device and printer device, a presentation space is displayed. Depending on the device context this presentation space is linked to, the output appears either in the window or on paper. The different aspects resulting from WYSIWYG display are demonstrated in PREVIEW.PRG.

Scaling

An automatic scaling of the drawing and the transformation of the coordinate systems of both output devices is done by the presentation space. It defines unit and size of the coordinate system. For printed output, the coordinate system must be dimensioned according to the paper size. It can be determined with the *:paperSize()* method of an XbpPrinter object. This method returns an array containing the size of a sheet of paper in 1/10th mm units. For some printers, especially laser printers, the size must be adjusted since there are margins that cannot be printed on.

Printer coordinates are used for drawing in the presentation space. This means that the page size of the presentation space is identical to the printable area on a sheet of paper. In order to display the complete page of the presentation space in the window, the viewport must be set to the window's size. In this case, window coordinates are used (pixel). If page size and viewport are dimensioned properly, all graphic output drawn at printer coordinates appears in correct scale in a window.

In addition to viewport and page size, it is necessary to select a vector font in the presentation space. This is required for correct scaling of text since bitmap fonts cannot be scaled. Therefore, a presentation space must be prepared in several steps to achieve a WYSIWYG preview of printed output. These steps are discussed in the following code (note: it is assumed that the variable *oPrinter* references an XbpPrinter object and *oWindow* references an Xbase Part):

```
// Determine page size
aSize := oPrinter:paperSize()

// Deduct margins that can not be printed on
aSize := { aSize[5]-aSize[3], aSize[6]-aSize[4] }

// Link presentation space with window device context
// but use printer coordinates and units
oPresSpace := XbpPresSpace():new()
oPresSpace:create( oWindow:winDevice(), ;
                   aSize              , ;
                   GRA_PU_LOMETRIC      )

// Set viewport to the size of the window
aSize := oWindow:currentSize()
oPresSpace:setViewPort( {0, 0, aSize[1], aSize[2] } )

// Select vector font for presentation space
oFont           := XbpFont():new( oPresSpace )
oFont:familyName := "Helvetica"
oFont:nominalPointSize := 12
oFont:create()
oPresSpace:setFont( oFont )
```

When a presentation space is prepared in this way, Gra..() functions can be used to produce graphic output. Display in the window then corresponds to printed output and appears in correct scale. However, screen output is extremely reduced in size compared to printed output.

Zooming

If details of a print preview are to be checked, the display in the window must be enlarged (zoomed). This is achieved by enlarging the viewport of the presentation space. In this case, only a part of the viewport is visible in the window. Assume the following code:

```
// Get current viewport size (e.g. { 0, 0, 270, 380 } )
aViewPort := oPresSpace:setViewPort()

// Double height and width of the viewport
// Its size is now 540 x 760 pixel
aViewPort[3] *= 2
aViewPort[4] *= 2

// Set new viewport in presentation space
oPresSpace:setViewPort( aViewPort )
```

The window is assumed to be 270 pixel wide and 380 pixel high. If the viewport is enlarged to double width and height (540 x 760 pixel), the window displays only the lower left quarter of the viewport. Since the window's size is unchanged, only one-quarter is displayed instead of the entire drawing. This part of the drawing appears four times enlarged.

Scrolling

As soon as the viewport is larger than the window, only a part of a drawing is visible and the rest is clipped. To see different parts of the drawing, it must be scrolled inside the window. Scrolling is done by changing the origin of the viewport relative to a window's origin. The origin of the viewport is set to negative coordinates:

```
// Get current viewport size (e.g. { 0, 0, 540, 760 } )
aViewPort := oPresSpace:setViewPort()
aViewPort[1] -= 270
aViewPort[2] -= 380
aViewPort[3] -= 270
aViewPort[4] -= 380

oPresSpace:setViewPort( aViewPort )
```

In this example, the viewport of the size 540 x 760 pixel is set to the coordinates {-270,-380,270,380}. This is relative to the point {0,0} in the window. The viewport origin has moved down left while its size is unchanged. As a result, the upper right part of the drawing becomes visible in the window.

If the viewport origin is set to negative coordinates, all parts of an enlarged drawing can be viewed in a window. In the PREVIEW.PRG example program, this is accomplished by using two scroll bars so that the drawing can be scrolled with mouse clicks.

Printing

When a presentation space is created using printer coordinates, a drawing visible on screen can be printed easily by exchanging the device context of the presentation space. For printing, an XbpPrinter object is used as device context and the drawing must be redisplayed for appearance on paper. The following steps are necessary to print a drawing:

```
// Save current viewport
aViewPort := oPresSpace:setViewPort()

// Link presentation space with printer device
oPresSpace:configure( oPrinter )

// Open the spooler
oPrinter:startDoc()

// Redisplay the drawing
Gra..( oPresSpace )

// Close the spooler
```

```
oPrinter:endDoc()

// Link presentation space to window device
oPresSpace:configure( oWindow:winDevice() )

// Reset viewport for window
oPresSpace:setViewPort( aViewPort )
```

The device context is exchanged by passing a corresponding object to the ***:configure()*** method of the presentation space. This switches graphic output from screen to the printer. Since the viewport is lost when devices are changed, it must be saved and reset.

18. Error Handling Concepts

In addition to code that addresses the problem that the program is designed to solve, code to handle error conditions is another important area of program development. In complex programs errors or exception conditions always occur, since no single programmer can test every possible combination in an extensive application, and no one can foresee every one of the large number of exception conditions which are possible. Potential error conditions must be considered when the application is being developed. Various strategies can be used in order to rectify potential errors within a program. There is no universal strategy. This chapter illustrates various approaches to planning for error conditions within a program during application development.

18.1. Offensive and defensive error handling

There are two "philosophies" in planning for handling error conditions during program development: one assumes that everything might be wrong and the other assumes that everything is right. These strategies have significant consequences on how procedures, user-defined functions, and methods are written, especially those that can receive arguments. Passing arguments to routines is a critical area for error handling because it frequently leads to errors, especially when individual arguments must have a specific data type or must lie within a specific range of values. A simple example is a user-defined function which performs a simple division operation.

```
FUNCTION Divide(nDividend, nDivisor)
RETURN  nDividend / nDivisor
```

This function presumes that both parameters are always of the numeric data type and that the parameter *nDivisor* is never equal to zero. If the programmer can guarantee these conditions, the function is acceptable as it is shown here. However, one must normally assume that incorrect arguments that could lead to a program error may be passed. Such a case can be avoided if the parameters are tested.

```
FUNCTION Divide(nDividend, nDivisor)
   LOCAL nResult := 0

   IF Valtype(nDividend) + Valtype(nDivisor)=="NN" .AND. ;
      nDivisor <> 0

      nResult := nDividend / nDivisor

   ENDIF
RETURN nResult
```

In this example, the function covers all possible error conditions. It tests whether both parameters are of numeric data type and avoids division by zero. In this form, the function *Divide()* follows the defensive strategy of error handling because the function avoids all potential sources of error. If an incorrect parameter is passed, the function simply substitutes a numeric result (the value zero).

The strategy of avoiding the source of all possible errors guarantees stable programs, but can lead to a significant reduction in the runtime performance of programs. In a carefully programmed application errors do not occur often. The frequent testing for potential errors is not generally required and these tests use significant time.

Thus, the defensive strategy of error handling has a negative effect on the runtime performance of a program. This is because tests are frequently executed that are seldom necessary and are often superfluous. It is more advantageous to use an offensive error handling strategy which avoids superfluous test routines and assumes that no error will occur. This philosophy is basic in a 32bit operating system and should also be followed when programming using Xbase^{++}.

The control structure BEGIN SEQUENCE...ENDSEQUENCE, in connection with the function ErrorBlock() and an error handling code block, form the basis for offensive error handling. The example function *Divide()* appears as follows using an offensive strategy:

```
FUNCTION Divide(nDividend, nDivisor)
   LOCAL nResult, bError

   bError := ErrorBlock( {|e| Break(e)} ) // install new error
                                          // handling code block
   BEGIN SEQUENCE
      nResult := nDividend / nDivisor     // normal program code
   RECOVER
      nResult := 0                        // error handling code
   ENDSEQUENCE

   ErrorBlock( bError )                   // re-install old error
                                          // handling code block
RETURN nResult
```

In this example, no test is performed on the passed parameters. Instead, it is assumed that the passed parameters always have the correct values and data types. The function contains two kinds of program code embedded in the BEGIN SEQUENCE ...ENDSEQUENCE structure. There is the program code which processes the normal, error free condition and additional program code that is executed if an error occurs. When an error occurs at runtime, the program code after the RECOVER statement is executed. Otherwise, this part of the function is skipped. In order for the RECOVER code to be executed, a local error handling code block is installed using the function ErrorBlock(). This code block calls the function Break() and is executed only if an error occurs. The function Break() interrupts the normal program execution and causes the program flow to continue after the RECOVER statement.

Although the function *Divide()* is not really needed, it demonstrates the two basic "philosophies" of runtime error handling. Offensive error treatment is generally advantageous since it provides improved runtime performance. For the programmer, offensive error handling means that, from inception of the development process, program code is organized into two parts. These parts are embedded in a BEGIN SEQUENCE...ENDSEQUENCE structure. One part contains the normal program logic. The other part captures error conditions and returns the program to a stable condition assuring a continued, uninterrupted run. Offensive error handling protects the main program code, which greatly improves the quality and maintainability of the program. All errors can be trapped with offensive error handling, which is not possible using the defensive strategy unless all possible conditions are explicitly tested. It should be noted that there is no universal rule to use in determining what program conditions are defined as errors and when simple error handling should be performed. The simple error conditions, like division by zero, are the exception rather than the rule. When in doubt, a subjective decision must be made concerning what constitutes an error and what form of error handling will be used. In many error conditions, it makes sense not to attempt error recovery, but to simply abort the program. In most error conditions, program termination is the default reaction of the error handling system of Xbase^{++} (see the file ERRORSYS.PRG).

18.2. Use of error objects

Objects containing information about runtime errors play a central role in the Xbase^{++} error handling system. Error objects are simple objects containing instance variables but no methods. An error object is automatically created when an error occurs and information about the error is contained in this object's instance variables. After an error object is created, it is passed to the error code block that was installed using the function ErrorBlock(). The error code block can call any functions or procedures and pass on to them the error object received by the code block. Using the function ErrorBlock(), user defined functions that use the information from an error object can be called to perform error handling when runtime errors occur.

In order to use the information in an error object, a parameter must be defined in the error code block. In case of an error, the parameter receives an error object that can be passed on to any function. The simplest case is passing the error object to the function Break() which continues the program flow after the RECOVER statement. This is implemented in the following program example. The program tests the operating readiness of drives. The test occurs in the user-defined function *IsDriveReady()* called from *Main()*. In the procedure *Main()*, a drive letter can be entered and this drive is tested for readiness.

```
**************
PROCEDURE Main                          // test drives for
   LOCAL cDrive := "C" , nReady         // readiness

   CLS                                  // clear screen
   DO WHILE LastKey() <> 27             // terminate with ESC key

      @ 0,0 SAY "Drive to test:" GET cDrive // enter drive
      READ
      nReady := IsDriveReady( cDrive ) // test drive

      @ 10,10                           // position cursor and
                                        // clear screen
      IF nReady == 0                    // display message
         @ 10,10 SAY  "Drive ready"
      ELSEIF nReady == -1
         @ 10,10 SAY  "Drive not ready"
      ELSE
         @ 10,10 SAY  "Drive not found or invalid"
      ENDIF
   ENDDO

RETURN

#define  DRIVE_NOT_READY    21          // OS error code


*******************************
FUNCTION IsDriveReady( cDrive )         // Is drive ready ?
   LOCAL nReturn   := 0
   LOCAL cOldDrive := CurDrive()        // save current drive
   LOCAL bError    := ErrorBlock( {|e| Break(e) } )
   LOCAL oError

   BEGIN SEQUENCE

      CurDrive( cDrive )                // change drive
      CurDir( cDrive )                  // read current directory

   RECOVER USING oError                 // error has occurred

      IF oError:osCode == DRIVE_NOT_READY
         nReturn := -1                  // drive not ready
      ELSE
         nReturn := -2                  // drive not available
      ENDIF                             // or invalid

   ENDSEQUENCE
   ErrorBlock( bError )                    // reset error code block
   CurDrive( cOldDrive )                   // and drive

RETURN nReturn
```

The user-defined function *IsDriveReady()* includes some important techniques for error handling. It starts by assigning the error code block {|e| Break(e) } which is executed if an error occurs. The code block passes an error object to the function Break(). For example, an error occurs when one of the characters "&/(123" is entered in *Main()* or when IsDriveReady("K") is called and drive "K" does not exist.

The function Break() continues the program flow after the RECOVER statement. In the example program, the RECOVER statement includes the USING option defining the variable *oError*. This variable receives the parameter passed to Break(), in this case it is the error object. The variable *oError* references an error object only after an error occurs. Otherwise it contains the value NIL and the program code between RECOVER and ENDSEQUENCE is not executed.

If an error occurs, an error object is automatically created and passed to the error code block. This error object contains information about the error in the values of its instance variables. Within the code block, the object is passed to the function Break() which branches to RECOVER. The error object is finally assigned to the variable *oError* specified in RECOVER USING. After an error occurs, the instance variables of the error object can then be accessed by the program code between RECOVER and ENDSEQUENCE. In the example, only the instance variable *:osCode* is inspected because it contains the error code for the errors that might occur during the test to determine the operating readiness of a drive (osCode = Operating System Code). Testing the drive is done by using the operating system which generates an error when a drive is not ready or does not exist. In the Xbase^{++} error handling system, the errors generated by the operating system are captured.

The function *IsDriveReady()* assigns the various return values in the error handling program code. This program code is not executed when the drive to be tested is ready. In this case, the essential program code contains only two lines. The function *IsDriveReady()* is a good example of the strategy of offensive error handling, demonstrates how errors can be captured, and shows how information about runtime errors can be used.

The most essential item for successful error handling is the error code block, which receives an error object. The error object contains information about the error that occurred. The program code for error handling appears between RECOVER and ENDSEQUENCE. The RECOVER statement occurs between BEGIN SEQUENCE and ENDSEQUENCE and can optionally receive a value as a parameter. The option RECOVER USING defines a variable to be assigned the value of the argument passed to Break().

18.3. Default error handling - ERRORSYS.PRG and XPPERROR.LOG

The default error handling of Xbase++ is implemented in the ERRORSYS.PRG file. It contains the function ErrorSys() which installs the default error code block at program startup. When a runtime error occurs, information about the error is displayed on screen. Depending on the error condition, the user can decide whether to ignore the error, retry the last operation or terminate the program.

In case of program temination an error log can be created optionally which gets written to the XPPERROR.LOG file. The error log contains important data about the error condition and provides the developer with valuable information. The situation when the error occured can be analyzed in detail which allows a programming error to be identified directly or helps to localize the error using the Xbase++ debugger. Information that may get listed in the error log is too numerous to describe. Therefore an example of an error log is discussed below. It explains how to interprete information contained in the log file. The example originates from the MDIDEMO sample program which is part of the Xbase++ installation.

```
------------------------------------------------------------
ERROR LOG

Xbase++ version      : Xbase++ (R) Version 1.10.153
Operating system     : Windows NT  4. 0 Build 01381
------------------------------------------------------------
oError:args          :
         -> VALTYPE: L VALUE: .T.
         -> VALTYPE: U VALUE: NIL
         -> VALTYPE: C VALUE: Customer
         -> VALTYPE: U VALUE: NIL
         -> VALTYPE: U VALUE: NIL
         -> VALTYPE: L VALUE: .F.
oError:canDefault    : .T.
oError:canRetry      : .T.
oError:canSubstitute: .F.
oError:cargo         : NIL
oError:description   : Operating system error
oError:filename      :
oError:genCode       :         40
oError:operation     : DbUseArea
oError:osCode        :          2
oError:severity      :          2
oError:subCode       :          4
oError:subSystem     : BASE
oError:thread        :          1
oError:tries         :          1
```

```
------------------------------------------------------------
CALLSTACK:
------------------------------------------------------------
Called from STANDARDEH(155)
Called from (B)ERRORSYS(24)
Called from OPENCUSTOMER(230)
Called from CUSTOMER(27)
Called from (B)MENUCREATE(28)
Called from MAIN(46)
```

The error log begins with version information about Xbase^{++} and the operating system. Two main sections follow: the contents of the error object and the call stack. The call stack lists the sequence of calls to functions, procedures or methods that has led to the error condition. The function name is followed by the line number of the PRG file where the function is called. In the example, the error log is initiated in line 155 of the *StandardEH()* function. This function is called from a code block which is programmed in line 24 of *ErrorSys()*. Both functions are part of the ERRORSYS.PRG file and implement the default error handling of Xbase^{++}. They do not contribute to a runtime error but are part of the call stack when the error log is created. The runtime error occured in line 230 of the *OpenCustomer()* function which is programmed in the MDICUST.PRG file. The call stack clearly identifies the program line that raises the runtime error. It cannot display the name of the PRG file, only line number and function name.

If the information contained in the error object is not sufficient to clearly identify a programming error, the error situation can be analyzed easily with the Xbase^{++} debugger. For this, the application must be started from the debugger and a break point must be set on line 230 of the MDICUST.PRG file.

In this example, however, the error object provides sufficient information to identify the error: the CUSTOMER.DBF file could not be found. The following code has raised the runtime error:

```
USE Customer NEW
```

This line of code is translated by the preprocessor and the actual code executed at runtime looks like this:

```
DbUseArea( .T., NIL, NIL, "Customer", .F. )
```

In its instance variable *:operation*, the error object contains a string with the name of the failed operation or function, respectively, and the instance variable *:args* contains all arguments or formal parameters passed to the function. The error log lists data types and values of the arguments (VALTYPE and VALUE). The instance variable *:description* gives a short description of the failed operation. In the example, the error is raised by the operating system, not by the Xbase^{++} application. This is also indicated by the instance variable *:osCode* which contains the value 2 in this case. If this instance variable contains a number not equal to zero, then this is an operating system error code.

Another important information is stored in the *:genCode* instance variable. It is the generic Xbase++ error code that corresponds to #define constants found in the ERRORSYS.CH file. The number 40 shown in the example equals to the constant XPP_ERR_DOS and indicates the error to be raised by the Disk Operating System.

Summary

The error log records the contents of an error object together with the call stack. The call stack indicates WHERE a runtime error occured while the error object provides information WHY it is raised. In many cases the contents of the instance variables *:arg*, *:description*, *:operation*, *:osCode* and *:genCode* are sufficient to clearly identify and resolve a runtime error (please refer to the reference documentation for a description of other instance variables of the error object).

Index