



DBMaker

DCI User's Guide



CASEMaker Inc./Corporate Headquarters
1680 Civic Center Drive
Santa Clara, CA 95050, U.S.A.
www.casemaker.com
www.casemaker.com/support

CASEMaker Inc./Asia Division
11th Floor, No. 260, Pa Teh Road, Section 2
Taipei, Taiwan, R.O.C.
Casemaker_Asia@email.syscom.com.tw

© Copyright 1995-2000 by CASEMaker Inc.
Document Nos.: UPC Code: 645049-290472, ISO Number: 1G1SRDP-DM371MU1

All rights reserved. No part of this manual may be reproduced, stored in a retrieval system, or transmitted in any form without the prior written permission of the manufacturer.

For a description of updated functions that do not appear in this manual, read the file named README.TXT after installing the CASEMaker DBMaker software.

Trademarks

CASEMaker, the CASEMaker logo, and DBMaker are registered trademarks of CASEMaker Inc. Microsoft, MS-DOS, Windows, and Windows NT are registered trademarks of Microsoft Corp. UNIX is a registered trademark of The Open Group. ANSI is a registered trademark of American National Standards Institute, Inc.

Other product names mentioned herein may be trademarks of their respective holders and are mentioned only for information purposes. SQL is an industry language and is not the property of any company or group of companies, or of any organization or group of organizations.

Notices

The software described in this manual is covered by the license agreement supplied with the software.

Contact your dealer for warranty details. Your dealer makes no representations or warranties with respect to the merchantability or fitness of this computer product for any particular purpose. Your dealer is not responsible for any damage caused to this computer product by external forces including sudden shock, excess heat, cold, or humidity, nor for any loss or damage caused by incorrect voltage or incompatible hardware and/or software.

Information in this manual has been carefully checked for reliability; however, no responsibility is assumed for inaccuracies. This manual is subject to change without notice.

Contents

1	Introduction	1-1
1.1	About this Manual	1-1
	Who Should Read This Book	1-1
	Information Contained in This Book	1-1
	Other Sources of Information	1-2
	Document Conventions.....	1-3
1.2	DCI Overview.....	1-3
	File System and Databases	1-5
	Accessing Data	1-6
2	DCI Getting Started.....	2-1
2.1	Technical Support:.....	2-1
2.2	Requirements	2-2
2.3	Setup Instructions	2-2
	Setup with Windows.....	2-2
	Setup with UNIX	2-4
2.4	Basic Configuration	2-7
	DCI-DATABASE.....	2-8
	DCI-LOGIN	2-8
	DCI-PASSWD	2-8
	DCI_XFDPATH.....	2-9
2.5	Sample Application	2-9

Setting up the Application	2-9
Adding Records	2-12
Accessing the Data.....	2-14

3 Data Dictionaries 3-1

3.1 Tables..... 3-1

3.2 Columns and Records 3-4

Identical Field Names.....	3-7
Long Field Names	3-7

3.3 Multiple Record Formats..... 3-8

3.4 Defaults Used in XFD Files..... 3-10

REDEFINES Clause	3-10
KEY IS phrase.....	3-11
FILLER data items	3-11
OCCURS Clauses	3-11

3.5 Mapping other files to an XFD 3-12

4 XFD Directives..... 4-1

4.1 Using Directives 4-1

4.2 Directive Syntax 4-1

4.3 XFD Directives Supported by DCI 4-2

\$XFD ALPHA Directive	4-2
\$XFD BINARY Directive	4-3
\$XFD COMMENT Directive	4-4
\$XFD DATE Directive	4-4
\$XFD FILE= <i>Filename</i> Directive.....	4-7
\$XFD NAME Directive	4-8
\$XFD NUMERIC Directive.....	4-8
\$XFD USE GROUP Directive	4-8
\$XFD VAR-LENGH Directive.....	4-9
\$XFD WHEN Directive	4-9

5	Invalid Data.....	5-1
6	ACUCOBOL-GT Compiler and Runtime Options.....	6-1
6.1	Filing System Options.....	6-1
6.2	Setting the Default Filing System	6-2
6.3	Setting the Filing System for Specific Files	6-2
6.4	Runtime Setting of Filing System	6-3
7	DCI Configuration File Variables.....	7-1
	DCI_DATABASE	7-2
	DCI_DATE_CUTOFF	7-2
	DCI_INV_DATA	7-3
	DCI_LOGFILE.....	7-3
	DCI_LOGIN	7-3
	DCI_JULIAN_BASE_DATE	7-4
	DCI_LOGTRACE.....	7-4
	DCI_MAPPING	7-4
	DCI_MAX_DATA.....	7-5
	DCI_MIN_DATA	7-5
	DCI_PASSWD.....	7-5
	DCI_STORAGE_CONVENTION.....	7-6
	DCI_USEDIR_LEVEL.....	7-7
	DCI_USER_PATH.....	7-7
	DCI_XFDPATH.....	7-8
	DEFAULT_RULES	7-8
	<i>filename_RULES(*)</i>	7-9
8	DCI Limit and Range	8-1
8.1	DCI Supported Features	8-1
8.2	Limits	8-1

8.3	Mapping COBOL Data Types to DBMaker Data Types.....	8-3
8.4	Mapping DBMaker Data Types to COBOL Data Types.....	8-4
8.5	Runtime Errors.....	8-7
8.6	Native SQL Errors	8-8
9	Converting ACUCOBOL Vision files ...	9-1
	Glossary.....	1
	Index.....	1

1 Introduction

1.1 About this Manual

This manual gives systematic instructions on how to use the DBMaker COBOL Interface (DCI), a program designed to allow for efficient management and integration of data with COBOL using the DBMaker database engine.

Who Should Read This Book

This book is intended for software developers who want to combine the reliability of COBOL programs with the flexibility and efficiency of a relational database management system (RDBMS). DCI provides a communication channel between your COBOL programs and the DBMaker RDBMS.

Information Contained in This Book

- *Getting Started* provides essential information for setting up and configuring DCI. It also provides information about running the demonstration program that will aid you in understanding the basic functions of DCI.
- *Data Dictionaries* describes how extended file descriptor (.XFD) files are created and accessed. How data is handled between DBMaker and COBOL programs is discussed in detail.
- Directives can be used to overcome problems such as incompatible data types and field names.

- *XFD Directives* introduces the concept of directives, and lists DCI compatible directives.
- *Invalid Data* lists data types that cannot be accepted by the RDBMS and lists solutions that DCI implements to solve this problem.
- *ACUCOBOL GT Compiler and Runtime Options* describes configuration settings for ACUCOBOL-GT that let you specify what filing system to use.
- *DCI Limit and Range* lists allowable ranges of data for DCI, as well as tables specifying how COBOL data types are mapped to DBMaker data types.
- *Converting ACUCOBOL Vision Files* gives detailed instructions on how to transfer data stored in a Vision file to the DBMaker RDBMS.

Other Sources of Information

DBMaker provides many other user's guides and reference manuals in addition to this guide. For more detailed information on a particular subject, you should consult one of the books listed below:

- *For more information on the SQL language used in dmSQL, refer to the "SQL Command and Function Reference".*
- *For more information on designing, administering, and maintaining a DBMaker database, refer to the "Database Administrator's Guide".*
- *For more information on the tools and utilities provided with DBMaker, refer to the "dmSQL User's Guide", the "DBATool User's Guide", or the "Server Manager User's Guide".*
- *For more information on error and warning messages, refer to the "Error and Message Reference".*
- *For more information on DBMaker configurations, functions, and management using the Java tools, refer to the "JConfiguration Tool Reference", the "JServer Manager User's Guide", or the "JDBA Tool User's Guide".*

Document Conventions

This book uses a standard set of typographic conventions for clarity and ease of understanding.

<i>Italics</i>	Italics indicate placeholders for information you must supply, such as usernames and table names. The word in italics should not be typed, but should be replaced by the actual name you want to use. Italics also introduce new words, and are occasionally used for emphasis in text.
Boldface	Boldface indicates filenames, database names, table names, column names, function names, and other similar terms. It is also used to emphasize menu commands in procedural steps.
<code>Computer</code>	Information that is displayed on the screen is presented in this type. This includes programming code, instructions that the product displays in response to user input, and instructions that you are required to enter.
KEYWORDS	All keywords used by the <i>SQL</i> language appear in uppercase when used in text.
SMALL CAPS	Small capital letters indicate keys on the keyboard. A plus sign (+) between two key names indicates you should hold down the first key while pressing the second. A comma (,) between two key names indicates you should release the first key before pressing the second key.

1.2 DCI Overview

DBMaker COBOL Interface (DCI) allows COBOL programs to efficiently access information stored in the DBMaker relational database. In order to store data, COBOL programs usually use standard B-TREE files. Information stored in B-TREE files are traditionally accessed through standard COBOL I/O statements, for example READ, WRITE and REWRITE.

COBOL programs can also access data stored in the DBMaker RDBMS. Normally, a technique called *embedded SQL* is used. COBOL programmers can embed SQL statements into the COBOL source code. Before compiling the source code, a special pre-compiler translates SQL statements into "calls" to the database engine. These calls are executed during the runtime in order to access the DBMaker RDBMS.

Though this technique is a good solution for storing information on a database using COBOL programs, it has some drawbacks. First, it implies COBOL programmers have a good knowledge of the SQL language. Second, a program written in this way is not portable. In other words, it cannot work both with B-TREE files and the DBMaker RDBMS. Furthermore, SQL syntax is often different from database to database. This means that a COBOL program embedding SQL statements tailored for a specific DBMaker RDBMS cannot work with a different database. Finally *embedded SQL* is difficult to implement with existing programs. In fact, *embedded SQL* requires significant application re-engineering, including substantial additions to working storage, data storage and a reworking of the logic of each I/O statement.

There is an alternative to embedded SQL. Some suppliers have developed seamless interfaces from COBOL to the database. These interfaces translate COBOL I/O commands into SQL statements on the fly. In this way, COBOL programmers need not be familiar with SQL and COBOL programs can stay portable. However, performance is the main problem here. In fact, SQL has a different purpose than COBOL I/O statements. SQL is intended to be a set-based, ad hoc query language that can find almost any combination of data from a general specification. By contrast, COBOL B-TREE (or other data structure) calls are designed for direct data access via well-defined traversal keys and/or navigation logic. Therefore, forcing transaction-rich, performance-sensitive COBOL applications to operate exclusively via SQL-based I/O is often an inappropriate method.

CASEMaker's COBOL interface product, DCI, does not use SQL for this reason. Instead, it provides for direct data storage access and traversal in a manner similar to the way COBOL itself accesses any other user-replaceable COBOL file system. DCI provides a seamless interface between your COBOL program and the DBMaker filing system. Information exchange between the application and the database are invisible to the end user. On the other hand, for desktop decision support systems (DSS), data

warehousing, or 4GL applications, DBMaker provides full SQL-based file/ data storage access as required, as well as the reliability and robustness of a RDBMS.

CASEMaker's Database and DCI products combine the power of 4GLs and navigational data structures with the ad hoc flexibility of SQL-based database access and reporting. They also provide startling performance.

File System and Databases

Although traditional COBOL file systems and databases both contain data, they differ significantly. Databases are generally more robust and reliable than traditional file systems. Furthermore, they act as efficient systems for data recovery from software or hardware crashes. In addition, in order to ensure data integrity, DBMaker RDBMS provides support for referential actions, as well as domain, column, and table constraints.

However, there are some parallels in the way data is stored by a database and COBOL indexed files. The following table shows the different data structures of each system and how they correspond to one another:

COBOL Indexed Filing System Object	Database Object
Directory	Database
File	Table
Record	Row
Field	Column

Database operations are performed on columns. Indexed file operations are performed on records.

A COBOL indexed file is logically represented in database table format. Within a table, each column represents one COBOL field and each row represents one COBOL record.

Database table columns are strictly associated with a particular data type, such as date, integer, or character. In COBOL, data can have multiple type definitions.

For example, a COBOL record that looks like this:

```
terms-record.  
      03      terms-code      pic 999.  
      03      terms-rate      pic s9v999.  
      03      terms-days      pic 9(2).  
      03      terms-descript  pic x(15).
```

would be represented in the database as a table with a format similar to this:

terms_code	terms_rate	terms_days	terms_descript
234	1.500	10	net 10
235	1.750	10	net 10
245	2.000	30	net 30
255	1.500	15	net 15
236	2.125	10	net 10
237	2.500	10	net 10
256	2.000	15	net 15

Each row is an instance of the 01 level record terms-record.

Accessing Data

ACUCOBOL-GT's generic file handler interfaces with DCI and the Vision filing system. Vision is the standard indexed file system supplied with ACUCOBOL-GT. Vision files are discussed in more detail in Chapter 11, "Converting ACUCOBOL Vision files".

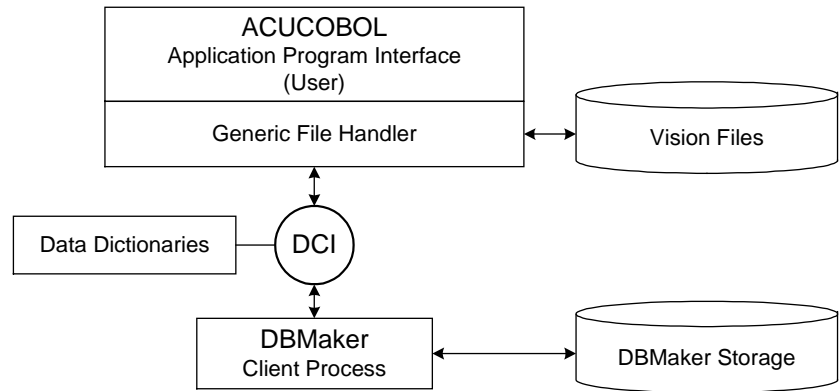


Figure 1-1 Flowchart of data

DCI, in combination with the data dictionaries, is a gateway between data access in a COBOL based application program interface (API) and the DBMaker data management system. Users may access data through the API. Furthermore, ad-hoc queries may be made on the data by using one of the DBMaker SQL interfaces: dmSQL, DBA Tool or JDBA Tool. Data dictionaries are created by the ACUCOBOL-GT compiler and are discussed in more detail in Chapter 4, *Data Dictionaries*.

2 DCI Getting Started

The “Getting Started” chapter will help you to install and set up DCI for DBMaker RDBMS. In this chapter you will find:

- Technical support information.
- Software and hardware requirements.
- Step-by-step setup instructions for UNIX and Windows.
- Options for configuring DCI.
- Instructions on how to view and use the DCI demonstration program.

2.1 Technical Support:

Please note that if you have any questions about DCI and DBMaker products you can contact us at:

CASEMaker Inc./Corporate Headquarters

1680 Civic Center Drive

Santa Clara, CA 95050, U.S.A.

www.casemaker.com, www.casemaker.com/support

Phone Number: +1-408-261-8265

Fax Number: +1-408-261-2153

E-mail: support@casemaker.com

2.2 Requirements

DCI for DBMaker is an add-on module that must be linked with the ACUCOBOL-GT runtime system. For this reason, you'll need a C compiler to install the DCI product. In order to interface, you must use an ACUCOBOL-GT Version 4.3 or later compiler and runtime.

The READ_ME.DCI file lists the files that are shipped with the product.

The following platforms are supported by DCI:

- SCO
- Solaris x86 2.7
- Win 32
- Linux 2.0

The following software must be installed for DCI to function:

- DBMaker 3.7 or later
- ACUCOBOL-GT runtime 4.3 or later
- A C compiler for the local machine (Visual C++™ Version 6.0 for a WINDOWS platform)

2.3 Setup Instructions

Setup with Windows

➤ To set up DCI:

1. Install DBMaker. DBMaker version 3.7 must be installed and configured prior to configuring DCI.

NOTE: Refer to the Quick Start insert included with the DBMaker CD for instructions on installation of DBMaker for Windows.

2. Start Windows.

3. Copy the DCI files from the source directory on the installation disk to your target directory.
4. Copy the DCI library file **dmdci37.lib** for windows to your local directory. For example:

```
Copy E:\dci\lib\win32\dmdci37.lib C:\local
```

5. Edit the ACUCOBOL runtime configuration file **filetbl.c**. It should be in the directory that contains the ACUCOBOL-GT libraries, for example: *c:\acucbl43\acugt\lib*.

The original **filetbl.c** contains the entry:

```
#ifndef USE_VISION
#define USE_VISION 1
#endif
```

Add a new entry as follows:

```
#ifndef USE_DCI
#define USE_DCI 1
#endif
```

The original **filetbl.c** contains the entry:

```
extern DISPATCH_TBL etc...;
```

Add a new entry as follows:

```
#if USE_DCI
extern DISPATCH_TBL DEM_dispatch;
#endif /* USE_DCI */
```

The original **filetbl.c** contains the entry:

```
TABLE_ENTRY file_table[] = {
    #if USE_VISION
        { &v4_dispatch, "VISIO" },
    #endif /* USE_VISION */
```

Add a new entry as follows:

```
#if USE_DCI
        { &DEM_dispatch, "DCI" },
    #endif /* USE_DCI */
```

6. Edit **wrun32.mak** by adding **dmdci37.lib** and **dmapi37.lib** to **CLIENT_LIBS**. It should be in the directory that contains the ACUCOBOL-GT libraries, for example: *c:\acucbl43\acugt\lib*.

Search for **CLIENT_LIBS** in **wrun32.mak** and add the following information:

```
CLIENT_LIBS=c:\dbmaker\3.7\lib\dmdci37.lib c:\dbmaker\3.7\lib\dmapi37.lib
```

7. Enter the command **nmake.exe -f wrun32.mak wrun32.exe** at the command prompt to build the **wrun32.exe** and **wrun32.dll**.
8. Copy the new **wrun32.exe** and **wrun32.dll** files to a directory mentioned in your execution path, for example *c:\acucbl43\acugt\bin*.
9. Verify the link: Type **wrun32 -vv** to verify the link. This will return version information on all of the products linked into your runtime system. Make sure it reports the version of the DBMaker interface.

Setup with UNIX

➞ To set up DCI:

1. Install DBMaker. DBMaker version 3.7 must be installed and configured before configuring DCI.

NOTE: Refer to the Quick Start insert included with the DBMaker CD for instructions on installation of DBMaker for UNIX.

2. Copy the DCI files from the source directory on the installation disk to your local directory.
3. Copy the DCI library file **libdmdcic.a** for UNIX to your local directory. For example:

```
cp /mnt/cdrom/dci/lib/linux/libdmdcic.a
```

4. Open the file Makefile. This should be located in *\usr\acucobol\43\lib*.
5. If you need to link in your own C routines, add them to a **SUBS=** line in the Makefile of your C routine. See Appendix C of the ACUCOBOL-GT compiler documentation for details on linking C subroutines.
6. Add **\$(DBMaker)/lib/libdmdcic.a** and **\$(DBMaker)/lib/libdmapic.a** to the line **FSI_LIBS=**, where **\$(DBMaker)** is the directory containing the DBMaker installation. For example, if DBMaker has been installed in the directory */DB/DBMaker* then the Makefile should contain the following string(s):

```
FSI_LIBS=/DB/DBMaker/lib/libdmdcic.a;  
/DB/DBMaker/lib/libdmapic.a
```

7. Next, make sure you are in the directory containing the ACUCOBOL-GT runtime system. At the UNIX prompt, type:

```
make -f Makefile <enter>
```

This will compile **sub.c** and **filetbl.c**, and will then link the runtime system.

NOTE: If the make fails because of an out-of-date symbol table, execute the following:

```
ranlib *.a <enter>
```

Then execute the make again; if the make fails for any other reason, call ACUCORP Technical Support.

8. Verify the link. Type:

```
./runcbl -vv
```

to verify the link. This will return version information on all of the products linked into your runtime system. Make sure it reports the version of DCI for DBMaker.

NOTES:

- *You must use Version 4.3 or greater of the ACUCOBOL-GT compiler and runtime in order to use the DBMaker product.*
- *You may also link your own C routines with the runtime system.*

9. Edit the ACUCOBOL runtime configuration file **filetbl.c**. It should be in the directory that contains the ACUCOBOL-GT libraries.

The original **filetbl.c** contains the entry:

```
#ifndef USE_VISION  
#define USE_VISION 1  
#endif
```

Add a new entry as follows:

```
#ifndef USE_DCI  
#define USE_DCI 1  
#endif
```

The original **filetbl.c** contains the entry:

```
extern DISPATCH_TBL etc...;
```

Add a new entry as follows:

```
#if USE_DCI
extern DISPATCH_TBL DBM_dispatch;
#endif /* USE_DCI */
```

The original **filetbl.c** contains the entry:

```
TABLE_ENTRY file_table[] = {
#if USE_VISION
    { &v4_dispatch, "VISIO" },
#endif /* USE_VISION */
```

Add a new entry as follows:

```
#if USE_DCI
    { &DBM_dispatch, "DCI" },
#endif /* USE_DCI */
```

- 10.** Copy the new **runcbl** file to a directory in your execution path. This file needs to have the execute permission for everyone who will be using the runtime system. The remaining files can be left in the directory into which they were installed from the distribution medium.

SHARED LIBRARIES

If you have relinked the ACUCOBOL-GT runtime and receive an error message of this type when you try to execute it:

"Could not load library; no such file or directory"

"Can't open shared library . . . "

This may mean that your operating system is using shared libraries and cannot find them. This can occur even if the shared libraries reside in the current directory.

Different versions of the UNIX operating system resolve this in different ways, so it is important that you consult your UNIX documentation to resolve this error. Some versions of UNIX require that you set an environment variable that points to shared libraries on your system. For example, on an IBM RS/6000 running AIX 4.1, the

environment variable LIBPATH must point to the directory where the shared libraries are located. On HP/UX, the environment variable that must be set to point to shared libraries is SHLIB_PATH. On UNIX SVR4, the environment variable is LD_LIBRARY_PATH. Be sure to read the system documentation for your operating system to determine the appropriate way to locate shared libraries.

A second way to resolve this type of error is to link the libraries into the runtime with a static link. Different versions of the C development system use different flags to accomplish this link. Please consult the documentation for your C compiler to determine the correct flag for your environment.

2.4 Basic Configuration

Two configuration files must have parameters set for DCI to work. The first is the ACUCOBOL runtime configuration file `filetbl.c` discussed in the setup procedure. The second is the DCI_CONFIG file that should be located in a directory determined by an environment variable (see “DCI Configuration File Variables” for details). To start working with DCI right away there are some important settings in the DCI_CONFIG file that need to be set. The DCI_CONFIG file sets parameters for DCI that determine how data appears in the database, as well as performs some basic DBA functions to allow access to the database. The following configuration variables need to be set in order to get DCI working.

- DCI_DATABASE
- DCI_LOGIN
- DCI_PASSWD
- DCI_XFDPATH

The following shows an example of a basic DCI_CONFIG file.

```
DCI_LOGIN SYSADM
DCI_PASSWD
DCI_DATABASE DBMaker_Test
DCI_XFDPATH /usr/DBMaker/Dictionaries
```

DCI-DATABASE

The database that all transactions from DCI are made to is specified by DCI-DATABASE. The database must first be established in the DBMaker setup. Note that database names are case-sensitive by default, and must be less than or equal to eighteen characters in length. If the database used is called

DBMaker_Test

the following entry must be included in the configuration file:

```
DCI-DATABASE DBMaker_Test
```

See “DCI_DATABASE” in chapter 7 for more information.

DCI-LOGIN

To ensure that your COBOL application has permission to access objects in the database, it should be given a username. The configuration variable DCI-LOGIN sets the username for all COBOL applications that use DCI. Initially this variable should be set to SYSADM to ensure full access to the database. This value can be set to another username. See “DCI_LOGIN” in chapter 7 for more information.

In order to connect to the database via the username SYSADM, the following must be specified in the DCI configuration file:

```
DCI-LOGIN SYSADM
```

DCI-PASSWD

Once a username has been specified via the DCI-LOGIN variable, a database account is associated with it. If the database account is set to SYSADM, then the configuration file should appear as follows:

```
DCI-PASSWD
```

Note there is no password for SYSADM. This is the default setting for DBMaker, but it can be changed. Consult with the database administrator to ensure that the account information (LOGIN, PASSWD) is correct. See “DCI_PASSWD” in chapter 7 for more information.

DCI_XFDPATH

DCI_XFDPATH is used to specify the name of the directory where data dictionaries are stored. The default value is the current directory. For example, if you want data dictionaries to be stored in the directory */usr/DBMaker/Dictionaries*, it is necessary to include the following entry in the configuration file:

```
DCI_XFDPATH /usr/DBMaker/Dictionaries
```

If it is necessary to specify more than one path, different directories have to be separated by spaces. For example:

```
DCI_XFDPATH /usr/DBMaker/Dictionaries /usr/DBMaker/Dictionaries1
```

In a WIN-32 environment you can specify “embedded spaces” by using double-quotes. For example:

```
DCI_XFDPATH c:\tmp\xfdlist "c:\my folder with space\xfdlist"
```

See “DCI_XFDPATH” in chapter 7 for more information.

2.5 Sample Application

Included in the DCI files is a small application program that will help you to understand how DCI maps application data to the DBMaker database. The following sections will tell you:

- How to set up the application program.
- How to compile the source code to create the application object code.
- How to input data to the application.
- How to access data using dmSQL and JDBC Tool.
- How the source code conforms to the schema of the generated table.

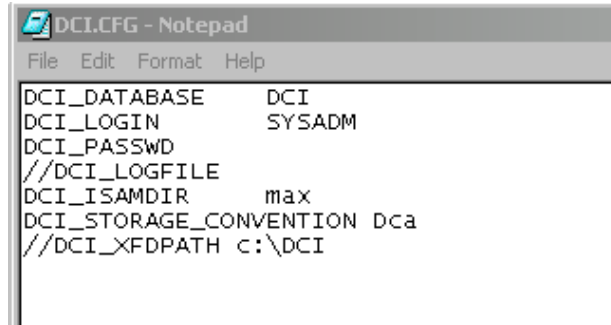
Setting up the Application

The application is located in the \DCI directory, and consists of the files INVD.CBL, INVD.FD, INVD.SL, TOTEM.DEF, and the object file INVD.ACU. The

application may be run directly from the object code (INVD.ACU) (see “To run the application”, below), or may be compiled from the source code (INVD.CBL) (see “To compile the source code”, below).

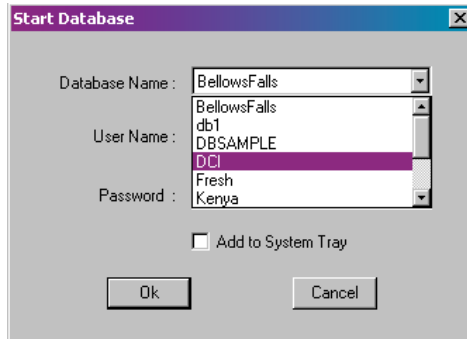
➡ To run the application:

1. Set up a database in DBMaker to accept data from DCI. As an example for this procedure, we created the database DCI using JServer Manager. All default settings were used for the database; specifically SYSADM was used for the default login name with no password. For information on creating and setting up a database, refer to the *Database Administrator's Reference* or the *JServer Manager User's Guide*.
2. In the \DCI directory, open the DCI.GFG file with any text editor. Set the configuration variables to appropriate values. Refer to section 2.6 “Basic Configuration” for information on configuration file values. In our example we use the following settings:

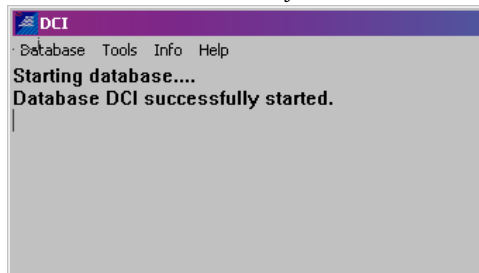


```
DCI_DATABASE      DCI
DCI_LOGIN         SYSADM
DCI_PASSWD
//DCI_LOGFILE
DCI_ISAMDIR       max
DCI_STORAGE_CONVENTION Dca
//DCI_XFDPATH     c:\DCI
```

3. Run the dmserver.exe program *included under the \DCI directory*. It will prompt you to start a database as shown below. Select the database that has been designated in the DCI.CFG file.



4. The database start normally and the following window will appear. If any problems or error messages occur, refer to the *Error Message Reference* or the *Database Administrator's Reference*.



5. Set the path for DCI from the command prompt. Open the command prompt and go to the \DCI directory. At the command prompt enter the following lines:

```
..\DCI>SET PATH=DCI
..\DCI>SET DCI_CONFIG=C:..\DCI\DCI.CFG
```

6. Run the COBOL file INVD.ACU. From the same directory at the command prompt, enter the following line:

```
..\DCI>WRUN32 -C CBLCONFIG INVD.ACU
```

7. The window of the COBOL application INVD.ACU will open as shown below, allowing you to enter values into the fields.

The screenshot shows a window titled "Screen" with a menu bar containing: First, review, Next, Last, a dropdown arrow, Refresh, Add, Update, Delete, Clear, and Exit. The main area contains the following fields:

INVD-INVLNO	<input type="text"/>
INVD-INVLSTKNO	<input type="text"/>
INVD-INVLDESC	<input type="text"/>
INVD-INVLQTY	<input type="text" value="0"/>
INVD-INVLFREE	<input type="text" value="0"/>
INVD-INVLPRICE	<input type="text" value="0000000.00"/>
INVD-MVTCODE	<input type="text"/>
INVD-SUBTOTAL	<input type="text" value="0000000.00"/>

For instructions on adding records, refer to “Adding Records” below.

☛ To compile the source code:

1. Follow steps 1 through 5 in “To run the application”, as above.
2. Copy the following definition files from the `..\Acucorp\Acucbl500\AcuGT\sample\def` directory to the `..\DCI` directory: `acucobol.def`, `acugui.def`, `crtvars.def`, `fonts.def`, `showmsg.def`.
3. At the command prompt go to the `..\DCI` directory. Enter the following line:

```
..\DCI>ccbl32 -Fx INVD.CBL
```
4. The file will be compiled and will create a new object code file `INVD.ACU`. To run the object file follow step 6 and 7 in “To run the application”, as above.

Adding Records

Once the application has been started (see “To run the application” above) it is a simple matter to add records to the application, and subsequently, to the database. The file descriptor for the application looks like this:

```
FD INVD
    LABEL RECORDS ARE STANDARD
    01      INVD-R
```

05	INVD-INVLNO	PIC X(10)
05	INVD-INVLSTKNO	PIC X(10)
05	INVD-INVLDESC	PIC X(30)
05	INVD-INVLQTY	PIC 9(8)
05	INVD-INVLFREE	PIC 9(8)
05	INVD-INVLPRICE	PIC 9(7)V99
05	INVD-MVTCODE	PIC X(6)
05	INVD-SUBTOTAL	PIC 9(7)V99

The field INVD-INVLNO is a key field, so a unique value is required for a record to be a valid entry. All other fields may be left blank. When you have finished entering values into the field, click on the **Add** button. The values you entered into the fields have now been saved in the DBMaker database specified by the DCI.CFG variable DCI-DATABASE. The following figure shows an example entry.

The screenshot shows a window titled "Screen" with a menu bar containing "First", "Previous", "Next", "Last", a drop-down menu, "Refresh", "Add", "Update", "Delete", "Clear", and "Exit". The main area contains a form with the following fields and values:

INVD-INVLNO	01
INVD-INVLSTKNO	
INVD-INVLDESC	Josef Albers
INVD-INVLQTY	1
INVD-INVLFREE	0
INVD-INVLPRICE	0004000.00
INVD-MVTCODE	
INVD-SUBTOTAL	0000000.00

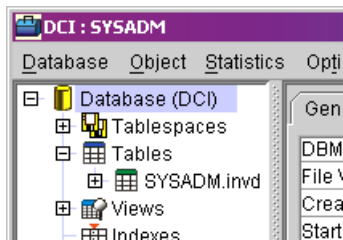
Once a record has been added, you may browse through the entries by selecting the **First**, **Previous**, **Next**, or **Last** buttons on the **Screen** window. An individual record may be selected from the drop-down menu at the top of the **Screen** window that

displays all the key field values. The section “Accessing the Data” describes how to browse data using DBMaker SQL based tools.

Accessing the Data

To browse and manipulate records created within the sample application is straightforward. First, we recommend that you familiarize yourself with one of the DBMaker tools: dmSQL, DBA Tool, or JDBC Tool. For information on the use of these tools, refer to the *Database Administrator's Guide*, the *DBA Tool User's Guide*, or the *JDBC Tool User's Guide*. The following example shows how data can be accessed using JDBC Tool.

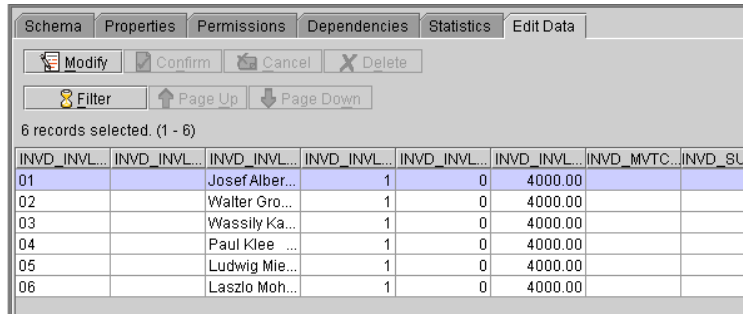
The INVD application must first be shut down, because it places a lock on the table that has been created within the database. Connect to the database with JDBC. You will be able to see the table by expanding the **Tables** node within the database tree as shown below.



Double clicking on the **SYSADM.invd** table will allow you to view the table's schema. All of the columns and their properties can be viewed here.

Schema Properties Permissions Dependencies Statistics Edit Data							
Modify Confirm Cancel							
Primary Key(s): INVD_INVLNO					X Del ▲ Up ▼ Down		
Name	Type	Precision	Scale	Nullable	Primary ...	Default V...	Constra
INVD_INVLNO	char	10		✓			
INVD_INVLSTK...	char	10		✓			
INVD_INVLDESC	char	30		✓			
INVD_INVLQTY	integer			✓			
INVD_INVLFREE	integer			✓			
INVD_INVLPRI...	decimal	7	2	✓			
INVD_MVTCODE	char	6		✓			
INVD_SUBTOT...	decimal	7	2	✓			

Selecting the **Edit Data** tab will allow you to view the values of each field as in the example below:



INVD_INVL...	INVD_INVL...	INVD_INVL...	INVD_INVL...	INVD_INVL...	INVD_INVL...	INVD_MVTC...	INVD_SU
01	Josef Alber...	1	0	4000.00			
02	Walter Gro...	1	0	4000.00			
03	Wassily Ka...	1	0	4000.00			
04	Paul Klee ...	1	0	4000.00			
05	Ludwig Mie...	1	0	4000.00			
06	Laszlo Moh...	1	0	4000.00			

3 Data Dictionaries

DCI avoids using SQL function calls imbedded in COBOL code by using a special feature of ACUCOBOL-GT. When a COBOL application is compiled using the `-Fx` option a set of indexed files are generated. These are known as eXtended File Descriptors (XFD files) or data dictionaries. DCI uses the data dictionaries to map data between the fields of a COBOL application and the columns of a DBMaker table. Every DBMaker table used by DCI has at least one corresponding data dictionary file associated with it.

NOTE: Please see ACUCOBOL-GT User's Guide (chapter 5.3) for information and rules concerning the creation of XFDs.

3.1 Tables

Database tables correspond to file descriptors in the FILE CONTROL section of the COBOL application. The database tables must have unique names under 18 bytes in length (18 ASCII characters). ACUCOBOL generates XFD file names by default from the FILE CONTROL section in the following ways:

ASSIGN *name* is a variable

If the SELECT statement for the file has a variable ASSIGN *name* (ASSIGN TO *filename*), then specify a starting name for the XFD file using a FILE directive (refer to \$XFD FILE=*Filename* Directive in chapter 4).

ASSIGN *name* is a constant

If the SELECT statement for the file has a constant ASSIGN *name* (such as ASSIGN TO “EMPLOYEE”), then the constant is used to generate the XFD file name.

ASSIGN *name* is generic

If the ASSIGN phrase refers to a device (such as ASSIGN TO “DISK”), then the compiler uses the SELECT name to generate the XFD file name.

File names are case-insensitive. All file descriptors containing upper case characters will be converted to lower case. Users must be aware of this if using a case sensitive operating system. For example, if the FILE CONTROL section contains the following:

```
SELECT FILENAME ASSIGN TO "Customer"
```

or

```
SELECT FILENAME ASSIGN TO "CUSTOMER"
```

DCI, based on dictionary information read in “customer.xfd”, will make a DBMaker table called “*username.customer*”. The Acucobol-GT compiler always creates a file name in lower case. Usernames are also case-insensitive.

The “username” default is determined by the DCI_LOGIN value in the DCI_CONFIG file, or can be changed with the DCI_USER_PATH configuration variable.

If the file has a file extension, DCI replaces “.” characters with “_”. Therefore, if you execute a statement like the following:

```
SELECT FILENAME ASSIGN TO "customer.dat"
```

DCI will open a DBMaker table named “*username.customer_dat*”. DCI_MAPPING can be used to make the dictionary customer.xfd available. Since DCI uses the base name to look for the XFD dictionary, in this case it looks for an XFD file named “*customer_dat.xfd*”. The following setting is based on an XFD file named “*customer.xfd*”

```
DCI_MAPPING customer*=customer
```

COBOL applications may use the same base file name in different directories. For example a COBOL application opens a file named “customer” in different directories such as “/usr/file/customer” and “/usr1/file/customer”. To make the file names unique we should include directory paths in the file names. A way to do this is to change the DCI_CONFIG variable DCI_USEDIR_LEVEL to “2”. DCI will then open a table as follows:

COBOL	RDBMS	XFD filename
/usr/file/customer	usrfilecustomer	usrfilecustomer.xfd
/usr1/file/customer	usr1filecustomer	usr1filecustomer.xfd

NOTE: Please remember there is a limit to the maximum length of DBMaker table names and that DCI_MAPPING must be used to map .XFD file dictionary definitions.

The following table gives examples of how table names are formed from different COBOL statements.

COBOL code	Resulting File Name	Resulting Column Name
ASSIGN TO “usr/hr/employees.dat	employees_dat.xfd	employees_dat
SELECT DATAFILE, ASSIGN TO DISK	datafile.xfd	datafile
ASSIGN TO “-D SYS\$LIB:EMP”	emp.xfd	emp
ASSIGN TO FILENAME	(user specified)	(user specified)

Table names are, in turn, generated from the XFD file name. Another way to specify the table name is to use the \$XFD FILE directive. Refer to \$XFD FILE=*Filename* Directive for details.

In summary, the final name is formed as follows:

- The compiler converts extensions and includes them with the starting name by replacing the “.” with an underscore.
- It constructs a universal base name from the file name and directory information as specified by the DCI_CONFIG variable DCI_USEDIR_LEVEL.

- It reduces the base name to eighteen characters and converts it to lower case.
- It adds the file extension “XFD”.

3.2 Columns and Records

The table that is created is based on the largest record in the COBOL file. It contains all the fields from that record and any key fields. Key fields are specified in the FILE CONTROL section using the KEY IS phrase. Key fields correspond to primary keys in the database table and are discussed in detail in the next section. Note that DCI will create column names for the database that are case sensitive, unlike table names.

The following example illustrates how data is transferred.

```
ENVIRONMENT DIVISION
INPUT-OUTPUT SECTION
FILE-CONTROL
    SELECT HR-FILE
        ORGANIZATION IS INDEXED
        RECORD KEY IS EMP-ID
        ACCESS MODE IS DYNAMIC

DATA DIVISION
FILE SECTION

FD HR-FILE
    LABEL RECORDS ARE STANDARD
01  EMPLOYEE-RECORD
    05  EMP-ID          PIC 9(6)
    05  EMP-NAME        PIC X(17)
    05  EMP_PHONE       PIC 9(10)

WORKING-STORAGE DIVISION
```

```
01      HR-NUMBER-FIELD

PROCEDURE DIVISION

PROGRAM-BEGIN

        OPEN I-O HR-FILE

        PERFORM GET-NEW-EMPLOYEE-ID

        PERFORM ADD-RECORDS

                UNTIL EMP-ID = ZEROES

        CLOSE HR-FILE

PROGRAM-DONE

        STOP RUN

GET-NEW-EMPLOYEE-ID

        PERFORM INIT-EMPLOYEE-RECORD

        PERFORM ENTER-EMPLOYEE-ID

INIT-EMPLOYEE-RECORD

        MOVE SPACE TO EMPLOYEE-RECORD

        MOVE ZEROES TO EMP-ID

ENTER-EMPLOYEE-ID

        DISPLAY "ENTER EMPLOYEE ID NUMBER (1-99999), ENTER 0 TO STOP ENTRY"

        ACCEPT HR-NUMBER-FIELD

        MOVE HR-NUMBER-FIELD TO EMP-ID

ADD-RECORDS

        ACCEPT EMP-NAME

        ACCEPT EMP PHONE

        WRITE EMPLOYEE-RECORD

        PERFORM GET-NEW-EMPLOYEE-NUMBER
```

The preceding program normally would write all fields sequentially to file. The output would appear as follows:

```
ENTER EMPLOYEE ID NUMBER (1-99999), ENTER 0 TO STOP ENTRY
51100
LAVERNE HENDERSON
2221212999
ENTER EMPLOYEE ID NUMBER (1-99999), ENTER 0 TO STOP ENTRY
52231
MATTHEW LEWIS
2225551212
ENTER EMPLOYEE ID NUMBER (1-99999), ENTER 0 TO STOP ENTRY
```

In a traditional COBOL filing system, records will be stored sequentially. Every time a write command is executed, the data is sent to the file. When DCI is used, the data dictionary will create a map for the data to be stored in the database. In this case, the record (EMPLOYEE-RECORD) is the only record in the file. The database will create a distinct column for each field in the file descriptor. The table name will be HR-RECORD in accordance with the SELECT statement in the FILE-CONTROL section. The database records in the example would therefore have the following structure:

Table EMPLOYEE-RECORD

EMP-ID (INT(5))	EMP-NAME (CHAR(17))	EMP-PHONE (DEC(10))
51100	LAVERNE HENDERSON	2221212999
52231	MATTHEW LEWIS	2225551212

In this table, the column EMP-ID is the primary key as defined by the KEY IS statement of the input-output section. The data dictionary creates a “map” that allows it to retrieve records and place them in the correct fields. A COBOL application that stores information in this way can take advantage of the backup and recovery features of the database, as well as take advantage of the capabilities of SQL.

Identical Field Names

In COBOL, fields with identical names are distinguished by qualifying them with a group item. In the following example you would reference PERSONNEL and PAYROLL in your program:

```
FD HR-FILE
    LABEL RECORDS ARE STANDARD
01  EMPLOYEE-RECORD
    03  PERSONNEL
        05  EMP-ID          PIC 9(6)
        05  EMP-NAME        PIC X(17)
        05  EMP_PHONE       PIC 9(10)
    03  PAYROLL
        05  EMP-ID          PIC 9(6)
        05  EMP-NAME        PIC X(17)
        05  EMP_PHONE       PIC 9(10)
```

DBMaker does not allow for duplicate column names on a table. If fields have the same name, DCI will not generate columns for those fields.

One solution to this situation is to add a NAME directive (Refer to \$XFD NAME Directive in chapter 4) that associates an alternate name with one or both of the conflicting fields.

Long Field Names

DBMaker allows for table names up to 18 characters in length. DCI will truncate field names longer than this. In the case of the OCCURS clause described below, the truncation is to the original name, not the appended index numbers. However, the final name, including the index number, is limited to the 18 characters. For example, if the field name is Employee-statistics-01 it would be truncated to form the table name Employee_statis_01. It is important to make sure that field names are unique (and meaningful) within the first 18 characters.

You can use the NAME to rename a field with a long name. Note that within the COBOL application you must continue to use the original name. The NAME directive affects only the corresponding column name in the database.

3.3 Multiple Record Formats

The example in the previous section shows how fields are used to create a database table. However, the example only shows the case of an application with one record.

A multiple record format will be stored differently from a single record format. COBOL programs with multiple records will map all records from the “master”(largest) record in the file and any key fields in the file. Smaller records are mapped to the database table by the XFD file but will not appear as discrete, defined columns in the table. Instead, they occupy new records in the existing columns of the database. Take the previous example but modify the file descriptor to include more than one record.

```
DATA DIVISION
FILE SECTION

FD HR-FILE
    LABEL RECORDS ARE STANDARD
01    EMPLOYEE-RECORD
        05    EMP-ID            PIC 9(6)
        05    EMP-NAME         PIC X(17)
        05    EMP PHONE        PIC 9(10)
01    PAYROLL-RECORD
        05    EMP-SALARY        PIC 9(10)
        05    DD                PIC 9(2)
        05    MM                PIC 9(2)
        05    YY                PIC 9(2)
```

In this case the data dictionary is created from the largest file. The record EMPLOYEE-RECORD contains 33 characters, while the record PAYROLL-

RECORD contains only 16. Records are entered sequentially into the database in this case. The record EMPLOYEE-RECORD is used to create the schema for the table: column size and data type. In the preceding example the table would look like this:

EMP-ID (INT(5))	EMP-NAME (CHAR(17))	EMP-PHONE (DEC(10))
-----------------	---------------------	---------------------

Fields from the following record would be written into the columns according to the character positions of the fields. The result is that no discrete columns exist for the smaller records. The data can be retrieved from the database by the COBOL application because the XFD file contains the map for the fields, but there are no columns in the table representing those fields.

In the previous example, when the first record is input into the database there is a correlation between the columns and the COBOL fields. When the second record is input there is no such correlation. The data occupies its corresponding character position according to the field. So the first five characters of EMP-SALARY occupy the EMP-ID column, the last five characters of EMP-SALARY occupy the EMP-NAME column. The fields DD and MM and YY are also located within the EMP-NAME column. The following example illustrates this.

Given the following input to the COBOL application:

```
ENTER EMPLOYEE ID NUMBER (1-99999), ENTER 0 TO STOP ENTRY
51100
LAVERNE HENDERSON
2221212999
5000000000
01
04
00
```

An SQL select on the first record of all columns in EMP-NAME would display the following:

```
51100, LAVERNE HENDERSON, 2221212999
```

An SQL select on the second record of all columns in EMP-NAME would display the following:


```
500000, 0000010400
```

The fields have been merged and split according to the character positions of the fields relative to the table's schema. Furthermore, the data type of the column EMP_NAME is CHAR. Because DCI has access to the data dictionary, all fields will be mapped back to the COBOL application in the correct positions.

This is a very important fact; by default, the fields of the largest record are used to create the schema of the table, therefore table schema must be carefully considered when creating file descriptors. To take advantage of the flexibility of SQL, data types should be consistent between fields for different records that will occupy the same character positions. If a PIC X field is written to a DECIMAL type database column, the database will return an error to the application.

3.4 Defaults Used in XFD Files

The compiler uses special methods to deal with the following COBOL elements:

- REDEFINES Clause
- KEY IS phrase
- FILLER data items
- OCCURS Clauses

This section describes in detail how the compiler deals with each element.

NOTE: In many cases you can use directives to override the default behavior of DCI. Refer to XFD Directives for more information.

REDEFINES Clause

A REDEFINES clause creates multiple definitions for the same field. DBMaker does not support more than one data definition per column. Therefore, a redefined field will occupy the same position in the table as the field that is being redefined. By default, the data dictionary uses the field definition of the subordinate field to define the column data type. The field being redefined is the field used to generate a column data type, not the new definition created after the REDEFINE clause.

Multiple record definitions are essentially redefines of the entire record area. Refer to the previous section for details on multiple record definitions.

Group items are not included in the data dictionary's definition of the resultant table's schema. Instead, the individual fields within the group item are used to generate the schema. Grouped fields may be combined using the USE GROUP directive.

KEY IS phrase

The KEY IS phrase in the input-output section of a COBOL program defines a field or group of fields as a unique index for all records. The data dictionary maps fields included in the KEY IS phrase to primary keys in the database. If the field named in the KEY IS phrase is a group item, the subordinate fields of the group item will become the primary key columns of the table. The USE GROUP directive can be employed to collect all subordinate fields into one field (see \$XFD USE GROUP Directive in Chapter 7).

FILLER data items

FILLER data items are placeholders in a COBOL file descriptor. They do not have unique names and cannot be uniquely referenced. The data dictionary maps all other named fields as if the fillers existed in terms of character position, but does not create a distinct field for the FILLER data item.

If a FILLER must be included in the table schema it can be combined with other fields using the USE GROUP directive (see \$XFD USE GROUP Directive in Chapter 7) or the NAME directive (see \$XFD NAME Directive in Chapter 7).

OCCURS Clauses

The OCCURS clause allows a field to be defined as many times as the user wants. DCI must assign a unique name for each database column, but multiple fields defined with an OCCURS clause will all have the same name. To avoid this problem, the field specified in the OCCURS clause is appended with a sequential index number.

For example, given the following part of a file descriptor:

```
03 EMPLOYEE-RECORD OCCURS 20 TIMES
```

05

EMP-ID

PIC 9(5)

The following column names would be generated for the database:

```
EMP_ID_1
EMP_ID_2
.
.
.
EMP_ID_5
EMP_ID_6
.
.
.
EMP_ID_19
EMP_ID_20
```

NOTE: Hyphens are converted to underscores in the database, and index numbers are preceded with an underscore.

3.5 Mapping other files to an XFD

At runtime, it is possible to use a single XFD file for files that have different names. For example, suppose a site has employee files that have identical structures but different names (EMP0001, EMP0002, EMP0003. etc.). If the record definitions are the same then it is not necessary to have a separate XFD for each file.

The individual files can all be mapped to the same XFD via a runtime configuration variable called DCI_MAPPING. The following paragraphs describe how it works.

Suppose your COBOL application has a SELECT with a variable ASSIGN name, such as employee-file. This variable assumes different values (such as EMP0001 and EMP0002) during program execution.

Before compiling the application, you would use the FILE directive (see \$XFD FILE=*Filename* Directive in Chapter 4) to provide a base name for the XFD. Suppose

you provide “EMP” as the base. The compiler would then generate an XFD named “emp.xfd”.

To ensure that all customer files, each having a unique but related name, will use the same XFD, make this entry in the runtime configuration file: DCI_MAPPING EMP* = EMP

The asterisk (“*”) in the example is a wildcard that matches any number of characters. Note that the file extension “.xfd” should not be included in the map. This statement would cause the XFD “emp.xfd” to be used for all files whose names begin with “EMP”.

The DCI_MAPPING variable has the following syntax:

DCI_MAPPING [*pattern* = base-xfd-name],

where *pattern* consists of any valid filename characters and may include “*” or “?”. These two characters have special meanings in the pattern:

* matches any number of characters

? matches a single occurrence of any character

For example:

EMP????? matches EMP00001 and EMPLOYEE, but does not match EMP001 or EMP0001

EMP* matches all of the above

EMP*1 matches EMP001, EMP0001, and EMP00001, but does not match EMPLOYEE.

*OYEE matches EMPLOYEE
does not match EMP0001 or EMP00001

The DCI_MAPPING variable is read during the open file stage.

4 XFD Directives

4.1 Using Directives

Directives are comments placed in your COBOL file descriptors that alter how the database table is built. Directives are used when a COBOL file descriptor is mapped to a database field. The \$XFD prefix indicates to the compiler that the following command is used in generation of the data dictionary.

Directives allow you to do the following:

- change the way data is defined in the database
- assign names to database fields
- assign file names to .XFD files
- assign data to binary large object (BLOB) fields
- add comments

4.2 Directive Syntax

Each directive should be placed on a line by itself, immediately before the related line of COBOL code. For example, the following command provides a unique database name for an undefined COBOL variable:

```
...  
          03      QTY          PIC 999  
01      CAP
```

```
$XFD NAME=CAPQTY  
      03      QTY      PIC 999
```

In the preceding example the directive is issued above the line it is intended to affect, in this case the second instance of the COBOL defined variable *qty*.

All directives have the prefix \$XFD; a \$ symbol in the 7th column followed immediately by XFD.

Alternatively, directives may be specified using the following ANSI-compliant syntax:

```
*(( XFD NAME=CAPQTY ))
```

Additionally, more than one directive may be combined. Directives should be on the same line, preceded by the prefix \$XFD and separated by a space or comma as in the following example:

```
$XFD NAME=CAPQTY, ALPHA
```

or

```
*(( XFD NAME=CAPQTY, ALPHA ))
```

4.3 XFD Directives Supported by DCI

\$XFD ALPHA Directive

If you want to store non-numeric data (for example, LOW-VALUES or special codes) in numeric keys, this directive allows a data item that has been defined as numeric in the COBOL program to be treated as alphanumeric text (CHAR (n) n 1-max column length) in the database.

The syntax for this directory is:

```
$XFD ALPHA
```

or

```
*(( XFD ALPHA ))
```

Examples:

Let's establish that the KEY IS code-key has been specified and we have this record definition:

```
01      code-record.
03      code-key.
        05      code-num      pic 9(5).
```

CODE-NUM is the key field here, since group items are disregarded in the database. Moving a non-numeric value such as "A234" to the key without using the \$XFD ALPHA directive would cause the record to be rejected by the database, since "A234" is an alphanumeric value and CODE-NUM is a numeric value.

By using the \$XFD ALPHA directive in this way,

```
01      code-record.
03      code-key.
$XFD ALPHA
        05      code-num      pic 9(5).
```

we can specify the following operation without worrying about rejection of the record:

```
MOVE "C0531" TO CODE-KEY.
WRITE CODE-RECORD.
```

\$XFD BINARY Directive

In order to allow for the date in a field to be alphanumeric data of any type (for example, LOW-VALUES), you can use the BINARY directive. In the case of LOW-VALUES, for example, COBOL allows both LOW and HIGH-VALUES in a numeric field, while DBMaker does not.

BINARY directives transform the COBOL fields into DBMaker BINARY data types.

The syntax for this directive is:

```
$XFD BINARY
```

or


```
*(( XFD BINARY ))
```

Example:

```
01      code-record.
03      code-key.
        05      code-indic          pic x.
*(( XFD BINARY ))
        05      code-num            pic 9(5).
        05      code-suffix        pic x(3).
```

This will allow LOW-VALUES to be moved to CODE-NUM

\$XFD COMMENT Directive

This directive lets you include comments in an XFD file. In this way, you can embed information in an XFD file so that other applications can access the data dictionary. Embedded information in the form of a comment using this directive does not interfere with processing by 4GL interfaces. Each comment will be recognizable in the XFD file as having the symbol “#” in column 1.

The syntax for this directive is:

```
$XFD COMMENT text
```

or

```
*(( XFD COMMENT text ))
```

\$XFD DATE Directive

DATE type data is a special data format supported by DBMaker that is not supported by COBOL. In order to take advantage of the properties of this data type fields must be converted from numeric type data. The DATE directive's purpose is to store a field in the database as a date. This directive differentiates dates from other numbers, so that they enjoy the properties associated with dates in the RDBMS.

The syntax for this directive is:

```
$XFD DATE=date-format-string
```

or

```
*((XFD DATE= ))
```

If no *date-format-string* is specified, then six-digit (or six-character) fields are retrieved as YYMMDD from the database. Eight-digit fields are retrieved as YYYYMMDD.

The *date-format-string* is a description of the desired date format, composed of characters from the following table:

M	Month (01-12)
Y	Year (2 or 4 digit)
D	Day of month (01-31)
J	Julian day (00000000-99999999)
E	Day of year (001-366)
H	Hour (00-23)
N	Minute (00-59)
S	Second (00-59)
T	Hundredths of a second

Each character in a date format string can be considered a placeholder that represents the type of information stored at that location. The characters also determine how many digits will be used for each type of data.

For example, although you would typically represent the month with two digits, if you specify MMM as part of your date format, the resulting date will use three digits for the month, with a left-zero filling the value. If the month is given as M, the resulting date will use a single digit, and will truncate on the left.

JULIAN DATES

Because the definition of Julian day varies, the DATE directive offers a great deal of flexibility for representing Julian dates, many source books define the Julian day as the

day of the year, with January 1st being 001, January 2nd being 002, and so forth. If you want to use this definition for Julian day, simply use FEE (day of year) in your date formats.

Other reference books define the Julian day as the number of days since some specific base date in the past. This definition is represented in the DATE directive with the letter J (for example, a six-digit date field would be preceded with the directive \$XFD DATE=JJJJJJ). The default base date for this form of Julian date is *01/01/0001AD*.

You may define your own base date for Julian date calculations by setting the configuration variable DCI-JULIAN-BASE-DATE.

DCI considers dates in the following range to be valid:

01/01/0001 to 12/31/9999

If a COBOL program attempts to write a record containing a date that DCI knows is invalid, DCI inserts a date value that depends on the setting specified by the DCI_INV_DATE, DCI_MIN_DATE, and DCI_MAX_DATE configuration variables into the date field and writes the record.

If a COBOL program attempts to insert into a record from a table with a NULL date field, zeroes are inserted into that field in the COBOL record.

If a date field has two-digit years, then years 0 through 19 are inserted as 2000 through 2019, and years 20 through 99 are inserted as 1920 through 1999. You can change this behavior by changing the value of the variable DCI-DATE-CUTOFF. Also, refer to the configuration variables DCI-MAX-DATE and A4GL-MIN-DATE for information regarding invalid dates when the date is in a key.

NOTE: If a field is used as part of a key, the field cannot be a NULL value.

USING GROUP ITEMS

You may place the DATE directive in front of a group item, so long as you also use the USE GROUP directive. For example:

```
$x fd date
      02      date-hired      pic 9(8).
```

```
03      pay-scale      pic x(11)
```

The column date-hired will have eight digits and will be type DATE in the database, with a format of YYYYMMDD. For example:

```
*(( XFD DATE, USE GROUP ))  
03      date-hired  
05      yyyy      pic      9(4).  
05      mm        pic      9(2).  
05      dd        pic      9(2).
```

\$XFD FILE=*Filename* Directive

The FILE directive names the data dictionary with the file extension .XFD. This directive is required when you want to make a different .XFD name from that specified in the SELECT COBOL statement.

Another case that requires this kind of directive is when the COBOL file name is not specific. For example, suppose you have the following COBOL syntax:

```
Select filename  
assign to variable-of-working.
```

You can use FILE directive in this way:

```
ENVIRONMENT DIVISION  
FILE-CONTROL  
Select filename  
assign to variable-of-working.  
  
DATA DIVISION  
FILE SECTION  
$XFD FILE=CUSTOMER  
fd filename
```

In this case, the ACUCOBOL-GT compiler makes an XFD file name called CUSTOMER.xfd.

\$XFD NAME Directive

The NAME directive assigns a DBMaker RDBMS column name to the field defined on the next line. In DBMS all field names should be unique and must be less than or equal to eighteen characters in length. You can use this directive to avoid problems created by fields with incompatible or duplicate names.

For example:

```
$xfd name=customercode  
03      cus-cod pic 9(5)
```

In DBMaker RDBMS, the COBOL field `cus-cod` will map to a RDBMS field named `customercode`.

\$XFD NUMERIC Directive

The NUMERIC directive causes the subsequent field to be treated as an unsigned integer if it is declared as alphanumeric.

For example:

```
$xfd numeric  
      03      customer-code      pic x(7).
```

The field `customer-code` will be stored as INTEGER type data in the DBMaker table.

\$XFD USE GROUP Directive

The USE GROUP directive assigns a group of items to a single column in the DBMaker table. The default data type for the resultant dataset in the database column is alphanumeric (CHAR (n) n 1-max column length). The directive may be combined with other directives if the data should be stored as a different type (BINARY, DATE, NUMERIC). The following example shows how the USE GROUP directive is used. Without this directive, data would be stored in the database as two numeric fields. By adding the directive, the data is stored as a single numeric field where the column name is *code-key*.

```
01      code-record.
```

```
$XFD USE GROUP
03      code-key.
        05      area-code-num          pic 9(3).
        05      code-num                pic 9(7)
```

Combining fields into groups improves processing speed on the database side, so effort should be made to determine which fields can be combined.

The USE GROUP directive can be combined with other directives. The following example shows how it can be used with the DATE directive.

```
*(( XFD USE GROUP, DATE ))
      03      date-hired
          05      yyyy      pic      9(4).
          05      mm       pic      9(2).
          05      dd       pic      9(2).
```

The fields above are mapped into a single DATE type data column in the database.

\$XFD VAR-LENGH Directive

VAR-LENGH directives force DBMaker to use a BLOB field to save COBOL fields. This is useful if the COBOL field is close to or above the maximum allowable column size for regular data types (refer to DCI Limit and Range).

Since BLOB fields cannot be used in any key field and are slower to retrieve than normal data type fields such as CHAR, we suggest you use this directive only when needed.

For example:

```
$XFD VAR-LENGH
03      large-field pic x(10000).
```

\$XFD WHEN Directive

The WHEN directive is used to build certain columns from the DBMaker RDBMS side that wouldn't normally be built by default. By specifying a WHEN directive in

the code, the field (and subordinate fields in the case of a group item) immediately following this directive will appear as an explicit column, or columns, in the database tables.

The database stores and retrieves all fields regardless of whether they are explicit or not. Furthermore, key fields and fields from the largest record automatically become explicit columns in the database table. The WHEN directive is only used to guarantee that additional fields will become explicit columns when you want to include multiple record definitions or REDEFINES in a database table.

One condition for how the columns are to be used is specified in the WHEN directive. Additional fields you want to become explicit columns in a database table must not be FILLER or occupy the same area as key fields.

The syntax for this directive is as follows:

```
$XFD WHEN field=value
```

(equals)

```
$XFD WHEN field<=value
```

(is less than or equals)

```
$XFD WHEN field<value
```

(is less than)

```
$XFD WHEN field>=value
```

(is greater than or equals)

```
$XFD WHEN field>value
```

(is greater than)

```
$XFD WHEN field!=value
```

(is not equal to)

```
$XFD WHEN field=OTHER
```

also:

```
*(( XFD WHEN field(operator)value ))
```

Where `value` is an explicit data value in quotes, and `field` is a previously defined COBOL field.

OTHER can only be used with the symbol “=”. In this case, the field or fields after OTHER must be used only if the WHEN condition or conditions listed at the same level are not met. OTHER can be used before one record definition, and, within each record definition, once at each level. It is necessary to use a WHEN directive with OTHER in the eventuality that the data in a field doesn’t meet the explicit conditions specified in the other WHEN directives. Otherwise, the results will be undefined.

Explicit data values in quotes (“”) are allowed.

Example:

```
03  AR-CODE-TYPE                                PIC X.
*(( XFD  WHEN AR-CODE-TYPE = "S"  ))
03  SHIP-CODE-RECORD                            pic x(4)
*(( XFD  WHEN AR-CODE-TYPE = "b"  ))
03  BACKORDER-CODE-RECORD REDEFINES SHIP-CODE-RECORD
*(( XFD  WHEN AR-CODE-TYPE = other ))
03  OBSOLETE-CODE-RECORD REDEFINES SHIP-CODE-RECORD
```

When you use tablename options in a WHEN statement, you must be aware of DEFAULT_RULES and filename_RULES DCI configuration variables. For example:

When you use tablename options in a WHEN statement, you must take care of DEFAULT_RULES and filename_RULES DCI configuration variables. For example:

If you have a FD structure like the following:

```
FD  invoice.
    $xfd when inv-type = "A" tablename=inv-top
    01  inv-record-top.
        03  inv-key.
            05  inv-type          pic x.
            05  inv-number       pic 9(5).
            05  inv-id           pic 999.
```



```
03 inv-customer      pic x(30).
$xfd when inv-type = "B" tablename=inv-details
01 inv-record-details.
03 inv-key-d.
05 inv-type-d        pic x.
05 inv-number-d       pic 9(5).
05 inv-id-b          pic 999.
03 inv-articles       pic x(30).
03 inv-qta           pic 9(5).
03 inv-price          pic 9(17).
```

The DCI interface makes two tables named “inv-top” and “inv-details” based on the value of the inv-type field. When DCI fills records, it checks the value of the inv-type field to know where to fill the record.

```
* make top row
    move "A" to inv-type
    move 1 to inv-number
    move 0 to inv-id.
    move "acme company" to inv-customer.
    write inv-record-top.

* make detail rows
    move "B" to inv-type
    move 1 to inv-number
    move 0 to inv-id.
    move "floppy disk" to inv-articles
    move 10 to inv-qta
    move 123 to inv-price
    write inv-record-details.
```

Running the above code, DCI fills the “top row” record in the “inv-top” table and “detail row” in the “inv-details” table. When DCI reads the above record, it can use sequential reading, or use the key to access filled records. If you plan to use sequential reading thru record types, you must set `DCI_DEFAULT_RULES = POST`. Alternately, if you plan to use sequential reading inside record types you must set `DCI_DEFAULT_RULES=BEFORE`.

There are advantages and disadvantages to using this rule. To have a 100% COBOL ANSI reading behavior, you should use the “POST” method, but this method can degrade performance (more records are read and all involved tables are open at the same time).

If you use the “BEFORE” method, the involved table is opened when the \$WHEN condition matches at the read record level.

In other words, if you use the previous records, and code the following statements:

```
open input invoice.
*   to see the custimer invoice
      read invoice next
      display "Customer  " inv-customer
      display "Invoice number "inv-number
*   to see the  invoice details
      read invoice next
      display inv-articles
```

If the method is “POST”, the “open input” opens both tables and “read next” reads thru different tables.

If the method is “BEFORE” the code should be changed as follows:

```
      open input invoice.
*   to see the customer invoice
      move "A" to  inv-type
      move 1   to  inv-number
      move 0   to  inv-id
```

```
        start invoice key is = inv-key.  
        read invoice next  
        display "Customer  " inv-customer  
display "Invoice number "inv-number  
*      to see the  invoice details  
        move "B" to  inv-type  
        move 1   to  inv-number  
        move 0   to  inv-id  
start invoice key is = inv-key.  
        read invoice next  
        display inv-articles
```

The matched table is opened at the “start” statement level.

5 Invalid Data

DCI uses some methods to manage data that is valid in COBOL applications but invalid for the DBMaker RDBMS database.

Here is a list of COBOL data that the database considers illegal:

Data Type	Where it is considered illegal
LOW-VALUES	In USAGE DISPLAY NUMBERS and text fields
HIGH-VALUES	In USAGE DISPLAY NUMBERS, COMP-2 numbers and COMP-3 numbers
SPACES	In USAGE DISPLAY NUMBERS and COMP-2 numbers
The value zero	In data fields

See the internal storage format of other numeric types to determine which of the above it applies to. Binary numbers are always legal, as are all values in binary text fields.

DBMaker does not accept some data types. DCI converts these values in the following ways:

- Illegal LOW-VALUES: stored as the lowest possible value (0 or - 99999...) or DCI-MIN-DATA default value.
- Illegal HIGH-VALUES: stored as the highest possible value (99999...) or DCI-MAX-DATA default value.

- Illegal SPACES: stored as zero (or DCI-MIN-DAT, in the case of a date field).
- Illegal DATE values: stored as DCI-INV-DATA default value.
- Illegal time: stored as DCI-INV-DATA default value.

Null fields sent to DCI from the database are converted to COBOL in the following ways:

- Numbers (including dates) are converted to zero.
- Text (including binary text) is converted to spaces.

6 ACUCOBOL-GT Compiler and Runtime Options

6.1 Filing System Options

Existing files opened with a COBOL application are associated with their respective filing systems as defined in the ACUCOBOL-GT configuration file. When new files are created by a COBOL application, you need to specify what filing system to use. The ACUCOBOL-GT configuration file needs to be set so that new files use the filing system of choice.

The following variables in the ACUCOBOL-GT configuration file allows you to use the filing system of choice:

```
DEFAULT-HOST (*)
```

and

```
filename-HOST (*)
```

The DEFAULT-HOST setting tells ACUCOBOL which filing system to use if no other system is specified for a new file. If no value has been given to this variable, ACUCOBOL will use the Vision file system as default. The filename-HOST setting allows you to set a filing system for a specific file. The name of the file should replace filename in the setting.

6.2 Setting the Default Filing System

In order to take advantage of DBMaker's reliability and features such as replication, backup and integrity constraints, we suggest using the DEFAULT-HOST DCI to avoid use of the ACUCOBOL-GT Vision filing system. The syntax is as follows:

```
DEFAULT-HOST DCI
```

In this case, all new files will be DBMaker files, unless the new files have been designated to a different filing system. In order to establish that all new files, unless otherwise specified, will be Vision files, the following should be used:

```
DEFAULT-HOST VISION
```

If no filing system is specified, the Vision filing system will be used by default.

6.3 Setting the Filing System for Specific Files

Filename-HOST is used to associate newly created files to a particular file system. It differs from the DEFAULT-HOST variable in that it associates single data files to a file system. In this way, you can have files that use a file system that is different from the one you have established as the default.

In order to do this, substitute the configuration file "DEFAULT" value, with the name of a file, without using directory names, or file extensions. DEFAULT-HOST and filename-HOST can be used together. Here is an example:

```
DEFAULT-HOST VISION  
file1-HOST DCI  
file2-HOST DCI
```

In this case, file 1 and 2 will use DBMaker, while the other files will use the Vision file system.

6.4 Runtime Setting of Filing System

In order to allow the file system to be set upon execution of your program, you could specify the following in your COBOL code:

```
SET ENVIRONMENT "filename-HOST" TO filesystem (*)
```

or

```
SET ENVIRONMENT "DEFAULT-HOST" TO filesystem (*)
```

(* Only for ACUCOBOL runtime)

Please be aware, however, that specification of a file system is usually done in the runtime configuration file and NOT changed in the COBOL program.

Please refer to the ACUCOBOL-GT, User's Manual (chapter 2.1 and 2.2) for instructions about how to use the ACUCOBOL-GT compiler and runtime.

7 DCI Configuration File Variables

Configuration file variables are used to modify the standard behavior of DCI and are stored in a file called DCI_CONFIG.

It is possible to give a configuration file a different address by setting a value to an environment variable called DCI_CONFIG. The value assignable to this environment variable can be either a full pathname or simply the directory where the configuration file resides. In this case, DCI will look for a file called DCI_CONFIG stored in the directory specified in the environmental variable. If the file specified in the configuration variable doesn't exist, DCI doesn't show an error and assumes no configuration variables have been assigned. This variable should be set in the COBOL runtime configuration file. See the ACUCOBOL-GT Compiler and Runtime Options for more information.

Example:

In Unix, working with Bourne shell, we can type the following command:

```
DCI_CONFIG=/usr/marc/config;export DCI_CONFIG
```

and DCI will look for the file DCI_CONFIG. This environment variable is used to establish the path and file configuration name of DCI.

Example:

In DOS:

```
set DCI_CONFIG=c:\etc\test
```

DCI reads the configuration file called DCI_CONFIG in the directory c:\etc\test

In UNIX:

```
DCI_CONFIG=/home/test/dci; export DCI_CONFIG
```

DCI utilizes the file called "DCI" in the directory /home/test

The following lists all the variables that can be included in the DCI_CONFIG file.

DCI_DATABASE

DCI_DATABASE is used to specify the name of the database that you have established in the DBMaker setup. If the database used is called

DBMaker_Test

the following entry has to be included in the configuration file:

```
DCI_DATABASE DBMaker_Test
```

Sometimes, the database name is not known in advance, and for this reason it is necessary to set it dynamically at the runtime. In cases like this, it is possible to write special code in the COBOL program similar to the one listed below.

The following code has to be executed before the first OPEN statement has been executed.

```
CALL "DCI_SETENV" USING "DCI-DATABASE" , "DBMaker_Test"
```

DCI_DATE_CUTOFF

This variable uses a two-digit value and establishes the two-digit years that will be interpreted by the program as being in the 20th Century and the two-digit years that will be interpreted by the program as being in the 21st Century.

The default value for the DCI_DATE_CUTOFF is 20. In this case, 2000 will be added to the two-digit years that are smaller than "20" (or whatever value you give to this variable), and will therefore make them part of the 21st Century. 1900 will be added to the two-digit years that are larger than "20" (or whatever value you give to this variable), making them part of the 20th Century.

Examples:

A COBOL date like 99/10/10 will be translated into 1999/10/10

A COBOL date like 00/02/12 will be translated into 2000/02/12

DCI_INV_DATA

This variable is used to establish an invalid date (like 2000/02/31) in order to avoid problems that can occur when invalid dates have been incorrectly written to the database.

The default for this variable is 99991230 (December 30th, 9999).

DCI_LOGFILE

This variable specifies the pathname of the log file used by DCI to write all the I/O operations executed by the interface. For example, by writing in the configuration file,

```
DCI_LOGFILE /tmp/dci_trace.log
```

the file *dci_trace.log*, stored in the directory */tmp*, is used as a log file. Log files are used for debugging purposes. The use of a log file slows down the performance of DCI. For this reason it is better not to include this variable in the configuration file (unless it is absolutely necessary).

DCI_LOGIN

DCI_LOGIN is a variable that allows for specification of a username in order to connect to the database system. It has no default value. Therefore, if no username is specified, no login will be used.

The username specified by the DCI_LOGIN variable should have RESOURCE authority or higher with the database. Additionally, the user should have permission with existing data tables. New users may be created using DBA Tool, JDBA Tool, or dmSQL. Refer to the *DBA Tool User's Guide*, *JDBA Tool User's Guide*, or the *Database Administrator's Guide* for detailed information on creating new users.

Example:

In order to connect to the database via the username JOHNDOE, one of the following must be specified in the DCI configuration file:

```
DCI_LOGIN JOHNDOE
```

DCI_JULIAN_BASE_DATE

This variable, used with the DATE directive, sets the base date for Julian date calculations. It utilizes the format YYYYMMDD. The default value for this variable is January 1st, 1 AD.

Example:

If your COBOL program uses dates from 1850 onwards, you can store these dates in a database by setting your DATE directive to \$YFD DATE=JJJJJJ (please note that your date field must have the same number of characters) and setting your DCI configuration variable DCI_JULIAN_BASE_DATE to 18500101.

DCI_LOGTRACE

This variable sets a different log level for tracing.

- 0: no trace
- 1: connect trace
- 2: record i/o trace
- 3: full trace
- 4: internal debug trace

DCI_MAPPING

This variable is used to associate particular filenames with a specific XFD in the DCI system. In this way, you can use one XFD in conjunction with multiple files.

The syntax for this variable is:

```
DCI_MAPPING [pattern = base-xfd-name] ...
```

DCI Configuration File Variables

A “*pattern*” can be made up of any valid filename characters. It may include the symbol “*”, which stands for any number of characters, or “?”, which stands for a single occurrence of any character.

Examples:

```
DCI_MAPPING CUST*1=CUSTOMER ORD*=ORDER "ord cli*=ordcli"
```

The pattern “CUST*1” and base-XFD-name “CUSTOMER” will cause filenames such as “CUST01”, “CUST001”, “CUST0001” and “CUST00001” to be associated with the XFD “customer.XFD”.

```
DCI_MAPPING CUST????=CUST
```

The pattern “CUST????” and base-XFD-name “CUST” will cause filenames such as “CUSTOMER” and “CUST0001” to be associated with the XFD “cust.XFD”.

DCI_MAX_DATA

This variable is used to establish a high-value date in order to avoid problems in cases where invalid dates have been incorrectly written to the database.

The default for this variable is 99991231 (December 31st, 9999).

DCI_MIN_DATA

This variable is used to establish a low-value, 0 or space date in order to avoid problems that can occur when invalid dates have been incorrectly written to the database.

The default for this variable is 00010101 (January 1st, 1AD).

DCI_PASSWD

Once a username has been specified via the DCI_LOGIN variable, a database account is associated with it. A password needs to be designated to this database account. This can be done using the variable DCI_PASSWD.

Examples:

If the password you want to designate to the database account is SUPERVISOR, the following must be specified in the configuration file:

```
DCI_PASSWD SUPERVISOR
```

A password can also be accepted from the user upon execution of the program. This allows for greater reliability. To do this, the DCI_PASSWD variable must be set according to the response:

```
ACCEPT RESPONSE NO-ECHO.  
CALL "DCI_SETENV" USING "DCI-PASSWD" , RESPONSE.
```

In this case, however, you should furnish a native API to call in order to read and write environment variables. For example:

```
CALL "DCI_SETENV", CALL "DCI_GETENV"
```

DCI_STORAGE_CONVENTION

This variable sets the storage convention. Possible values:

DCI

Selects the IBM storage convention. It is compatible with IBM COBOL, as well as with several other COBOL versions including RM/COBOL-85. It is also compatible with the X/Open COBOL standard.

DCM

Selects the Micro Focus storage convention. It is compatible with Micro Focus COBOL when the Micro Focus "ASCII" sign-storage option is used (this is the Micro Focus default).

DCN

Causes a different numeric format to be used. The format is the same as the one used when the "-DCI" option is used, except that positive COMP-3 items use "x0B" as the positive sign value instead of "x0C". This option is compatible with NCR COBOL.

DCA

Selects the ACUCOBOL-GT storage convention. It is the default setting. This convention is also compatible with data produced by RM/COBOL (not RM/COBOL-85) and previous versions of ACUCOBOL-GT.

DCI_USEDIR_LEVEL

If this variable is set > 0, use the directory in addition to the name of the table.

Examples:

```
DCI_USEDIR_LEVEL 1
```

```
/usr/test/01/clients          01clients
```

```
DCI_USEDIR_LEVEL 2
```

```
/usr/test/01/clients          test01clients
```

```
DCI_USEDIR_LEVEL 3
```

```
/usr/test/01/clients          usrtest01clients
```

DCI_USER_PATH

When the DCI looks for a file or files, the variable DCI_USER_PATH allows for specification of a username, or names, to do this. Here is an example of the syntax:

```
DCI_USER_PATH user1 [user2] [user3] .
```

The results of this setting will be determined by the kind of OPEN issued for a file. The user argument can be a period (.) with regard to your own files, or the name of a user on the system.

Examples:

Type of OPEN Issued	DCI_USER_PATH defined in configuration file	Sequence of places DCI Searches for a user	Result
1) OPEN INPUT or OPEN I/O	Yes	1-list of users in USER PATH 2-the current user	The first valid file will be opened.

Type of OPEN Issued	DCI_USER_PATH defined in configuration file	Sequence of places DCI Searches for a user	Result
2) OPEN INPUT or OPEN I/O	No	The user associated with DCI-LOGIN.	The first file with a valid user/file- name will be opened.
3) OPEN OUTPUT	Yes or no	Doesn't search for a user.	A new table will be made for the name associated with DCI-LOGIN.

DCI_XFDPATH

DCI_XFDPATH is used to specify the name of the directory where data dictionaries are stored. The default value is the current directory. For example, if you want data dictionaries to be stored in the directory */usr/DBMaker/Dictionaries*, it is necessary to include the following entry in the configuration file:

```
DCI_XFDPATH /usr/DBMaker/Dictionaries
```

If it is necessary to specify more than one path, different directories have to be separated by spaces. For example:

```
DCI_XFDPATH /usr/DBMaker/Dictionaries /usr/DBMaker/Dictionaries1
```

In a WIN-32 environment, you can specify "embedded spaces" using double-quotes. For example:

```
DCI_XFDPATH c:\tmp\xfdlist "c:\my folder with space\xfdlist"
```

DEFAULT_RULES

Default management method for the WHEN directive in multi-definition files.

Possible values:

POST

BEFORE

POST means the table is open when the \$WHEN condition is matched

BEFORE means that all related table are open when the COBOL application opens multi-definition files.

filename_RULES

Default management for a multi-definition file named “*filename*”.

For example, all files use the POST rule except the file CLIENT:

DEFAULT_RULES	POST
CLIENT_RULES	BEFORE

This command is only used for ACUCOBOL runtime.

8 DCI Limit and Range

8.1 DCI Supported Features

- OPEN ALLOWING READERS is not supported by the DCI. Transactions are enforced in DCI. All I/O operations are done using transactions. DCI sets autocommit off and manages DBMaker transactions to make record changes for users available.
- DCI fully supports COBOL transaction statements like START TRANSACTION, COMMIT/ROLLBACK TRANSACTION.

NOTES:

- *DCI doesn't support record encryption, record compression, or alternate collating sequence. If you have these options in your code, they will be disregarded.*
- *DCI doesn't support "P" picture edit in the XFD data definition.*
- *DCI doesn't support case-sensitive file-names. All file names are converted to lower case.*

8.2 Limits

There are certain limits DCI has that COBOL doesn't have. They are:

- Maximum indexed key size: 1024

- Maximum number of columns per key: 16
- Maximum number of columns: 252
- Maximum length of a CHAR field: 3992 bytes
- Maximum simultaneous DBMS connections: 240
- Maximum character limit on column names: 18

Furthermore, when using DCI, a single process can open a maximum of 255 database tables simultaneously.

DBMaker can use the same sort or retrieval sequence as the Vision file system, but it requires that a BINARY directive be put before each key field containing signed numeric data. High and low values can create complications in key fields.

The maximum precision for decimal fields is 17. If a variable must have 18 digit precision, you must use XFD ALPHA to change the field type definition.

The following DBMaker data types are not currently supported:

- OID
- SERIAL(Start)
- VARCHAR(size)
- FILE

The following DBMaker data types are currently supported using special directives:

DATE	Using XFD DATE
TIME	Using XFD DATE
TIMESTAMP	Using XFD DATE
LONGVARCHAR	Using XFD VAR-LENGH
LONGVARBINARY	Using XFD VAR-LENGH*
BINARY	Using XFD BINARY

*NOTE: *a special change in the .XFD file is needed to do this.*

8.3 Mapping COBOL Data Types to DBMaker Data Types

DCI establishes what it considers to be the best match of COBOL data types in the creation of any and all columns of a DBMaker database table. Any data the COBOL date type can contain can be contained in the database column and will be as close as possible to the type of data used in the COBOL program. In general, the XFD directives which have been specified are checked first, so that when, for example, the data should be of a binary type, a binary type in the database is located and used.

COBOL	DBMaker
9(1-4)	SMALLINT
9(5-9)	INTEGER
9(10-17)	DECIMAL(10-17)
s9(1-4)	SMALLINT
s9(5-9)	INTEGER
s9(10-17)	DECIMAL(10-17)
9(n) comp-1 n (1-17)	INTEGER
s9(n) comp-1 n (1-17)	INTEGER
9(1-4) comp-2	SMALLINT
9(5-9) comp-2	INTEGER
9(10-17) comp-2	DECIMAL(10-17)
s9(1-4) comp-2	SMALLINT
s9(5-9) comp-2	INTEGER
s9(10-17) comp-2	DECIMAL(10-17)
9(1-4) comp-3	SMALLINT
9(5-9) comp-3	INTEGER
9(10-17) comp-3	DECIMAL(10-17)
s9(1-4) comp-3	SMALLINT
s9(5-9) comp-3	INTEGER
s9(10-17) comp-3	DECIMAL(10-17)
9(1-4) comp-4	SMALLINT

COBOL	DBMaker
9(5-9) comp-4	INTEGER
9(10-17) comp-4	DECIMAL(10-17)
9(1-4) comp-5	SMALLINT
9(5-17) comp-5	DECIMAL(10)
s9(1-4) comp-5	SMALLINT
s9(5-17) comp-5	DECIMAL(10)
9(1-4) comp-6	SMALLINT
9(5-9) comp-6	INTEGER
9(10-17) comp-6	DECIMAL(10-17)
s9(1-4) comp-6	SMALLINT
s9(5-9) comp-6	INTEGER
s9(10-17) comp-6	DECIMAL(10-17)
signed-short	INTEGER
unsigned-short	INTEGER
signed-int	CHAR(10)
unsigned-int	CHAR(10)
signed-long	CHAR(18)
unsigned-long	CHAR(18)
float	FLOAT
double	DOUBLE
pic x(n)	CHAR(n) n 1- max column length

8.4 Mapping DBMaker Data Types to COBOL Data Types

DCI reads data from the database by doing a COBOL-like MOVE from the native data types (most of which have a CHAR representation so you can display them by using dmSQL) to the COBOL data types.

It is not necessary to worry about matching the database data types and COBOL data types exactly. PIC X(nn) can be used for each column with regards to database types having a CHAR representation. PIC 9(9) is a closer COBOL match for databases that have INTEGER types. The more you know about a database type, the more flexible you can be in finding a matching COBOL type. For example, if a column in a DBMaker database only contains values between zero and 99 (0-99), PIC 99 would be a sufficient COBOL date match.

Choosing COMP-types can be left to the discretion of the programmer since it has little effect on the COBOL data used. BINARY data types will usually be re-written without change, because they are foreign to COBOL. However, a closer analysis of BINARY columns might allow you to find a different solution. The DECIMAL, NUMERIC, DATE and TIMESTAMP types have no exact COBOL matches. They are returned from the database in character form, so the best COBOL data type equivalent would be USAGE DISPLAY.

The following table illustrates the best matches for database data types and COBOL data types:

DBMaker	COBOL
SMALLINT	9(1-4)
INTEGER	9(5-9)
DECIMAL(10-17)	9(10-17)
SMALLINT	s9(1-4)
INTEGER	s9(5-9)
DECIMAL(10-17)	s9(10-17)
INTEGER	9(n) comp-1 n (1-17)
INTEGER	s9(n) comp-1 n (1-17)
SMALLINT	9(1-4) comp-2
INTEGER	9(5-9) comp-2
DECIMAL(10-17)	9(10-17) comp-2
SMALLINT	s9(1-4) comp-2
INTEGER	s9(5-9) comp-2
DECIMAL(10-17)	s9(10-17) comp-2

DBMaker	COBOL
SMALLINT	9(1-4) comp-3
INTEGER	9(5-9) comp-3
DECIMAL(10-17)	9(10-17) comp-3
SMALLINT	s9(1-4) comp-3
INTEGER	s9(5-9) comp-3
DECIMAL(10-17)	s9(10-17) comp-3
SMALLINT	9(1-4) comp-4
INTEGER	9(5-9) comp-4
DECIMAL(10-17)	9(10-17) comp-4
SMALLINT	9(1-4) comp-5
DECIMAL(10)	9(5-17) comp-5
SMALLINT	s9(1-4) comp-5
DECIMAL(10)	s9(5-17) comp-5
SMALLINT	9(1-4) comp-6
INTEGER	9(5-9) comp-6
DECIMAL(10-17)	9(10-17) comp-6
SMALLINT	s9(1-4) comp-6
INTEGER	s9(5-9) comp-6
DECIMAL(10-17)	s9(10-17) comp-6
INTEGER	signed-short
INTEGER	unsigned-short
CHAR(10)	signed-int
CHAR(10)	unsigned-int
CHAR(18)	signed-long
CHAR(18)	unsigned-long
FLOAT	float
DOUBLE	double
CHAR(n) n 1-max column length	pic x(n)

8.5 Runtime Errors

Runtime errors have the format “9D, xx”, where “9D” indicates a file system error (reported in the FILE STATUS variable) and “xx” indicates a secondary error code.

The following is a list of DCI secondary errors:

Number	Definition	Interpretation	Solution
9D,01	There is a read error on the dictionary file.	An error occurred while reading the XFD file, XFD file is corrupt.	Recompile with -Fx to re-create the dictionary file.
9D,02	There is a corrupt dictionary file and cannot be read.	The dictionary file for a COBOL file is corrupt.	Recompile with -Fx to re-create the dictionary file.
9D,03	A dictionary file (.xfd) has not been found.	The dictionary file for a COBOL file cannot be found.	Specify correct directory in DCI_XFDPATH config. file variable(it may be necessary to recompile with -Fx).
9D,04	There are too many fields in the key.	There are more than 16 fields in a key.	Check key definitions, re-structure illegal key, recompile with -Fx.
9D,12	There is an unexpected error on a DBMaker library function.	A DBMaker library function returned an unexpected error.	
9D,13	The size for the “xxx” variable is illegal.	An elementary data item in your FD is larger than 255 bytes.	
9D,13	The type of data for the “xxx” variable is illegal.	There is no DBMaker type that matches the data type used.	
9D,14	There is more than one table with the same name.	More than one table had the same name when they were listed.	

8.6 Native SQL Errors

Some native SQL errors may be generated by your database while using DCI for DBMaker. The exact error number and wording may vary, according to your database. Here is a list of them:

Number	Definition	Interpretation	Solution
9D,????	There are too many BLOBs(binary large objects).	Some databases impose restrictions on how many BLOBs can be used in A single table.	
9D,????	Invalid column name or reserved word.	A column was named using a word that has been reserved for the database.	Compare a file trace of CREATE TABLE to the list of database reserved words. Apply NAME directive to the FD field of invalid column and recompile to create a new XFD file.

9 Converting ACUCOBOL Vision files

DCI provides a sample program to convert COBOL files to DBMS tables. This program is a general-purpose program that converts any Vision file to a DBMaker table.

Before using this program you need the following:

- A Vision file to convert.
- An XFD data dictionary for Vision file.
- The ACUCOBOL runtime system 4.3 or higher linked to DCI.
- A DCIMIGRATE object program.

You can use this command in the following way:

```
runcbl DCIMIGRATE vision_file_name dbm_table_name
```

where *vision_file_name* is the name of a vision file to convert and *dbm_table_name* is the name of a DBMaker table.

To run well you must define the minimum DCI configuration settings to work with DBMaker (DCI_LOGIN, DCI_DATABASE, DCI_PASSWD etc) and match the .XFD file name with *dbm_table_name* or use DCI_MAPPING to specify the name and location

The program DCIMIGRATE reads vision files and writes DBMaker tuples through DCI. In addition, after migration, it checks if all records are correct by reading vision records and comparing them with reading DBMaker rows.

The program DCIMIGRATE, will report the following information:

- Total record read successful
- Total record write successful
- Total record read unsuccessful
- Total record write unsuccessful
- Total record compared successful
- Total record compared unsuccessful

You can avoid this report by setting the environment variable named DCI_MIGRATE to “yes”. The report will then append a file named “dbm_table_name.log”.

```
DCI_MIGRATE = yesg
```

You can dump the record of an “unsuccessful operation” by adding “dump” to the DCI_MIGRATE setting. (Spaces will be considered separators. Log file names with embedded spaces are not permitted)

```
DCI_MIGRATE = yes dump
```

Glossary

API

Application Programming Interface. The API is a the interface between an application and it's operating system or other services.

Binary large Object (BLOB)

A large block of data stored in the database that is not stored as distinct records in a table. A BLOB cannot be accessed through the database as ordinary records can. The database only can access the name and location of a BLOB. Typically, another application is used to read the data.

Buffer

An internal memory space where data is temporarily stored during input or output operations.

Client

A computer that accesses and manipulates data that is stored on a central server.

Column

A set of data in a database table defined as multiple records of the same data type.

Data dictionaries

Also known as extended file descriptors. They serve as maps between database schema and the file descriptors of a COBOL application.

Directive

An optional comment placed in the COBOL code that sets the following field or fields to a data type other than the default DCI setting.

Field

A part of a COBOL file descriptor roughly corresponding to a database column. It is a discrete data item contained in a COBOL record.

File Descriptor

A file descriptor is an integer that identifies a file that is operated on by a process. Operations that read, write, or close a file use the file descriptor as an input parameter.

Indexed file

A file containing a list of keys that identify each record uniquely.

Key

A unique value used to identify a record in a database. See primary key.

Primary key

A column consisting of unique (or key) values that can be used to identify individual records in a table.

Query

A user's request for information. In DBMaker, SQL commands are used to execute data queries.

Record

In COBOL, a group of related fields defined in the Data Division. In DBMaker, a record is also referred to as a row, and defines a set of related data items in the columns of a table.

Relational Database

A database system where tables internal to one database as well as tables on different databases may be related to one another by using keys or unique indexes.

Schema

The structure of a database table as defined by its columns. Data type, size, number of columns, keys, and constraints all define a table's schema.

Server

A central computer that stores and distributes data for access by clients.

SQL

Structured Query Language. The language by which DBMaker and other ODBC compliant programs access data.

Table

A logical storage unit in a database that consists of columns and records (or rows).

XFD file

An acronym for extended file descriptor or data dictionary. It also forms the file extension for the data dictionary.

Index

A

ALPHA Directive, 4-2

ASSIGN *name*, 3-1

B

BINARY Directive, 4-3

B-TREE

Files, 1-3

C

CHAR field

Maximum Length, 8-2

Column Names

Maximum Length, 8-2

Columns, 3-4

Maximum Number, 8-1

COMMENT Directive, 4-4

Configuration

Basic, 2-7

Configuration file variables, 7-1

Configuration File Variables

_DCI_MAPPING, 7-4

CDI_LOGFILE, 7-3

DCI_DATABASE, 7-2

DCI_DATE_CUTOFF, 7-2

DCI_INV_DATA, 7-3

DCI_JULIAN_BASE_DATE, 7-4

DCI_LOGIN, 7-3

DCI_LOGTRACE, 7-4

DCI_MAX_DATA, 7-5

DCI_MIN_DATA, 7-5

DCI_PASSWD, 7-5

DCI_STORAGE_CONVENTION, 7-6

DCI_USEDIR_LEVEL, 7-7

DCI_USER_PATH, 7-7

DCI_XFDPATH, 7-8

DEFAULT_RULES, 7-8

*filename*_RULES, 7-9

Connections

Maximum Number, 8-2

D

Data Dictionaries, 3-1

Storage Location, 7-8

Data Structures, 1-5

Data Types

COBOL to DBMaker, 8-3

- DBMaker to COBOL, 8-4
- Not Supported, 8-2
- Supported, 8-2
- Database Name
 - Specifying, 7-2
- DATE Directive, 4-4
- DCI_CONFIG, 7-1
- DCI_DATABASE, 7-2
- DCI_DATE_CUTOFF, 7-2
- DCI_INV_DATA, 7-3
- DCI_JULIAN_BASE_DATE, 7-4
- DCI_LOGFILE, 7-3
- DCI_LOGIN, 7-3
- DCI_LOGTRACE, 7-4
- DCI_MAPPING, 3-12, 7-4
- DCI_MAX_DATA, 7-5
- DCI_MIN_DATA, 7-5
- DCI_PASSWD, 7-5
- DCI_STORAGE_CONVENTION, 7-6
- DCI_USEDIR_LEVEL, 7-7
- DCI_USER_PATH, 7-7
- DCI_XFDPATH, 7-8
- Default Filing System, 6-2
- DEFAULT_RULES, 7-8
- DEFAULT-HOST setting, 6-1
- Directives, 4-1
 - ALPHA, 4-2
 - BINARY, 4-3
 - COMMENT, 4-4
 - DATE, 4-4
 - FILE, 4-7
 - NAME, 4-8
 - NUMERIC, 4-8

- Supported, 4-2
- Syntax, 4-1
- USE GROUP, 4-8
- VAR-LENGH, 4-9
- WHEN, 4-9
- Document Conventions, 1-3

E

- embedded SQL*, 1-4
- Errors
 - Runtime, 8-7
 - SQL, 8-8
- Extended File Descriptors, 3-1

F

- Field Names
 - Identical, 3-7
 - Long, 3-7
- FILE CONTROL section, 3-1
- FILE Directive, 4-7
- File System, 1-5
- FILE=*Filename* Directive, 4-7
- filename_RULES(*)*, 7-9
- Filing System Options, 6-1
- FILLER data items, 3-11

I

- I/O Statements, 1-3
- Illegal DATE values, 5-2
- Illegal HIGH-VALUES, 5-1
- Illegal LOW-VALUES, 5-1
- Illegal SPACES, 5-2
- Illegal time, 5-2

Invalid Data, 5-1

J

Julian dates, 4-5

K

Key fields, 3-4

KEY IS phrase, 3-4, 3-11

L

Limits, 8-1

Login, 7-3

M

Multiple Record Formats, 3-8

N

NAME Directive, 4-8

NUMERIC Directive, 4-8

O

OCCURS Clauses, 3-11

P

Password, 7-5

Platforms

 Supported, 2-2

Primary Keys, 3-4

R

Records, 3-4

REDEFINES Clause, 3-10

Requirements

 Software, 2-2

 System, 2-2

Runtime Configuration File, 2-3

Runtime Errors, 8-7

Runtime Options, 6-1

S

Sample Application, 2-10

Schema, 3-10

SELECT statement, 3-1, 3-6

Setup, 2-2

 UNIX, 2-4

 Windows, 2-2

Shared Libraries, 2-6

Software Requirements, 2-2

Sources of Information, 1-2

SQL

 Embedded, 1-4

 Errors, 8-8

Supported Features, 8-1

Supported Platforms, 2-2

System Requirements, 2-2

T

Table Schema, 3-10

Tables, 3-1

Technical Support:, 2-1

U

USE GROUP Directive, 4-8

Username, 7-3

V

VAR-LENGTH Directive, 4-9

Vision file system, 6-1

W

WHEN Directive, 4-9

X

XFD files, 3-1