



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

©Copyright 1995-2006 by CASEMaker Inc.

Document No. 45049-231307/DBM43-M01202006-DCIU

Publication Date: 2006-01-20

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 Introduction1-1**
 - 1.1 Additional Resources1-3**
 - 1.2 Technical Support.....1-4**
 - 1.3 Document Conventions.....1-5**
- 2 DCI Basics2-1**
 - 2.1 DCI Overview2-2**
 - File System and Databases 2-2
 - Accessing Data 2-3
 - 2.2 System Requirements2-5**
 - 2.3 Setup Instructions.....2-6**
 - Setup with Windows 2-6
 - Setup with UNIX..... 2-9
 - 2.4 Basic Configuration2-14**
 - DCI_DATABASE 2-14
 - DCI_LOGIN 2-15
 - DCI_PASSWD 2-15
 - DCI_XFDPATH..... 2-15
 - 2.5 The Runsql Utility.....2-17**
 - 2.6 Invalid Data2-18**
 - 2.7 Sample Application.....2-20**
 - Setting up the Application..... 2-20

	Adding Records	2-23
	Accessing the Data	2-24
3	Data Dictionaries	3-1
3.1	Assigning Table Names	3-2
3.2	Mapping Columns and Records	3-5
	Identical Field Names	3-7
	Long Field Names	3-8
3.3	Using Multiple Record Formats.....	3-9
3.4	Using XFD File Defaults	3-12
	REDEFINES Clause	3-12
	KEY IS Phrase	3-12
	FILLER Data Items	3-13
	OCCURS Clauses	3-13
3.5	Mapping Multiple Files	3-14
3.6	Mapping to Multiple Databases	3-16
3.7	Using Triggers.....	3-20
3.8	Using Views.....	3-22
3.9	Using Synonyms.....	3-25
3.10	Open Tables in Remote Databases.....	3-26
3.11	Using DCI_WHERE_CONSTRAINT.....	3-28
4	XFD Directives	4-1
4.1	Using Directive Syntax	4-2
4.2	Using XFD Directives.....	4-3
	\$XFD ALPHA Directive	4-3
	\$XFD BINARY Directive	4-4
	\$XFD COMMENT DCI SERIAL n Directive.....	4-4
	\$XFD COMMENT DCI COBTRIGGER Directive.....	4-5
	\$XFD COMMENT Directive.....	4-5
	\$XFD DATE Directive.....	4-6
	\$XFD FILE Directive.....	4-9

	\$XFD NAME Directive.....	4-9
	\$XFD NUMERIC Directive.....	4-10
	\$XFD USE GROUP Directive	4-10
	\$XFD VAR-LENGTH Directive	4-11
	\$XFD WHEN Directive for File Names.....	4-12
	\$XFD COMMENT DCI SPLIT.....	4-16
5	Compiler and Runtime Options.....	5-1
5.1	Using ACUCOBOL-GT Default File System	5-2
5.2	Using DCI Default File System.....	5-3
5.3	Using Multiple File Systems.....	5-4
5.4	Using the Environment Variable.....	5-5
6	Configuration File Variables	6-1
6.1	Setting DCI_CONFIG Variables.....	6-2
	DCI_CASE.....	6-2
	DCI_COMMIT_COUNT.....	6-3
	DCI_DATABASE	6-3
	DCI_DATE_CUTOFF	6-4
	DCI_DEFAULT_RULES.....	6-5
	DCI_DEFAULT_TABLESPACE.....	6-5
	DCI_DUPLICATE_CONNECTION	6-5
	DCI_GET_EDGE_DATES.....	6-5
	DCI_INV_DATE	6-6
	DCI_LOGFILE.....	6-6
	DCI_LOGIN	6-6
	DCI_JULIAN_BASE_DATE.....	6-7
	DCI_LOGTRACE.....	6-7
	DCI_MAPPING	6-7
	DCI_MAX_ATTRS_PER_TABLE.....	6-8
	DCI_MAX_BUFFER_LENGTH.....	6-9
	DCI_MAX_DATE	6-9
	DCI_MIN_DATE	6-9

DCI_NULL_ON_ILLEGAL_DATE.....	6-9
DCI_PASSWD.....	6-10
DCI_STORAGE_CONVENTION.....	6-11
DCI_USEDIR_LEVEL.....	6-11
DCI_USER_PATH.....	6-12
DCI_XFDPATH.....	6-13
<filename>_RULES.....	6-13
DCI TABLE CACHE Variables	6-13
DCI_TABLESPACE.....	6-14
DCI_AUTOMATIC_SCHEMA_ADJUST.....	6-15
DCI_INCLUDE.....	6-15
DCI_IGNORE_MAX_BUFFER_LENGTH.....	6-15
DCI_NULL_DATE.....	6-15
DCI_NULL_ON_MIN_DATE	6-16
DCI_DB_MAP	6-16
DCI_VARCHAR.....	6-16
DCI_GRANT_ON_OUTPUT.....	6-16

7 DCI Functions 7-1

7.1 Calling DCI functions..... 7-2

DCI_SETENV.....	7-2
DCI_GETENV	7-2
DCI_DISCONNECT	7-2
DCI_GET_TABLE_NAME	7-3
DCI_SET_TABLE_CACHE	7-3
DCI_BLOB_ERROR.....	7-4
DCI_BLOB_GET.....	7-4
DCI_BLOB_PUT	7-6
DCI_GET_TABLE_SERIAL_VALUE	7-7
DCI_FREE_XFD	7-8

8 COBOL Conversions..... 8-1

8.1 Using Special Directives 8-2

8.2 Mapping COBOL Data Types 8-3

8.3 Mapping DBMaker Data Types.....8-5

8.4 Troubleshooting Runtime Errors.....8-7

8.5 Troubleshooting Native SQL Errors.....8-9

8.6 Converting Vision Files8-12

 Using DCI_Migrate..... 8-12

GlossaryGlossary-1

IndexIndex-1

1 Introduction

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). The 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.

DCI provides a communication channel between COBOL programs and DBMaker. 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 like READ, WRITE and REWRITE.

COBOL programs can also access data stored in the DBMaker RDBMS.

Traditionally, COBOL programmers use a technique called embedded SQL to 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 often varies from database to database. This means that a COBOL program embedding SQL

statements tailored for a specific DBMaker RDBMS cannot work with another database. Finally embedded SQL is difficult to implement with existing programs. In fact, embedded SQL requires significant application re-engineering, including substantial additions to the working storage, data storage, and reworking 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 a COBOL program and the DBMaker file 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.

1.1 Additional Resources

DBMaker provides a complete set of DBMS manuals in addition to this one. For more detailed information on a particular subject, consult one of the books listed below:

- For an introduction to *DBMaker*'s capabilities and functions, refer to the "*DBMaker Tutorial*".
- For more information on designing, administering, and maintaining a *DBMaker* database, refer to the "*Database Administrator's Guide*".
- For more information on *DBMaker* management, refer to the "*JServer Manager User's Guide*".
- For more information on *DBMaker* configurations, refer to the "*JConfiguration Tool Reference*".
- For more information on *DBMaker* functions, refer to the "*JDBA Tool User's Guide*".
- For more information on the *dmSQL* interface tool, refer to the "*dmSQL User's Guide*".
- For more information on the SQL language used in *dmSQL*, refer to the "*SQL Command and Function Reference*".
- For more information on the *ESQL/C* programming, refer to the "*ESQL/C User's Guide*".
- For more information on the native ODBC API, refer to the "*ODBC Programmer's Guide*".
- For more information on error and warning messages, refer to the "*Error and Message Reference*".

1.2 Technical Support

CASEMaker provides thirty days of complimentary email and phone support during the evaluation period. When software is registered an additional thirty days of support will be included. Thus, extending the total support period for software to sixty days. However, CASEMaker will continue to provide email support for any bugs reported after the complimentary support or registered support has expired (free of charges).

Additional support is available beyond the sixty days for most products and may be purchased for twenty percent of the retail price of the product. Please contact sales@casemaker.com for more details and prices.

CASEMaker support contact information for your area (by snail mail, phone, or email) can be located at: www.casemaker.com/support. It is recommended that the current database of FAQ's be searched before contacting CASEMaker support staff.

Please have the following information available when phoning support for a troubleshooting enquiry or include the information with a snail mail or email enquiry:

- Product name and version number
- Registration number
- Registered customer name and address
- Supplier/distributor where product was purchased
- Platform and computer system configuration
- Specific action(s) performed before error(s) occurred
- Error message and number, if any
- Any additional information deemed pertinent

1.3 Document Conventions

This book uses a standard set of typographical conventions for clarity and ease of use. The NOTE, Procedure, Example, and CommandLine conventions also have a second setting used with indentation.

CONVENTION	DESCRIPTION
Italics	Italics indicate placeholders for information that must be supplied, such as user and table names. The word in italics should not be typed, but is replaced by the actual name. Italics also introduce new words, and are occasionally used for emphasis in text.
Boldface	Boldface indicates filenames, database names, table names, column names, user names, and other database schema objects. It is also used to emphasize menu commands in procedural steps.
KEYWORDS	All keywords used by the SQL language appear in uppercase when used in normal paragraph text.
small caps	Small capital letters indicate keys on the keyboard. A plus sign (+) between two key names indicates to hold down the first key while pressing the second. A comma (,) between two key names indicates to release the first key before pressing the second key.
NOTE	Contains important information.
➔ Procedure	Indicates that procedural steps or sequential items will follow. Many tasks are described using this format to provide a logical sequence of steps for the user to follow
➔ □ Example	Examples are given to clarify descriptions, and commonly include text, as it will appear on the screen.
CommandLine	Indicates text, as it should appear on a text delimited screen. This format is commonly used to show input and output for dmSQL commands or the content in the dmconfig.ini file

Figure 1-1 Document Conventions Table

2 DCI Basics

This chapter provides essential information pertaining to setting up and configuring a DCI environment for DBMaker. It also provides information on running the demonstration program that assists in understanding the basic functions of DCI.

The following topics are covered in this chapter:

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

2.1 DCI Overview

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.

File System and Databases

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 FILE SYSTEM OBJECT	DATABASE OBJECT
Directory	Database
File	Table
Record	Row
Field	Column

Figure 2-1 COBOL and Database Object Structures

Indexed file operations are performed on records in COBOL and operations are performed on columns in a database. Logically, a COBOL indexed file represents a database table. Each record in a COBOL file represents a table row in a database and each field represents a table column. Data can have multiple definition types in COBOL while table columns in a database have to be associated with a particular data type such as integer, character, or date.

➤ Example

A COBOL record is defined using the following format:

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

The COBOL record displayed in the above example would be represented in a database as shown below. Each row is an instance of the COBOL 01 level record terms-record.

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

Figure 2-2 COBOL Records Converted to Database Rows

Accessing Data

ACUCOBOL-GT's generic file handler interfaces with DCI and the Vision file system. Vision is the standard indexed file system supplied with ACUCOBOL-GT. Vision files are discussed in more detail in Chapter 9.

DCI, in combination with the data dictionaries, is a gateway between data access in a COBOL based application program interface (API) and the DBMaker database 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 or JDBC Tool. Data dictionaries are created by the ACUCOBOL-GT compiler and are discussed in more detail in Chapter 3, *Data Dictionaries*.

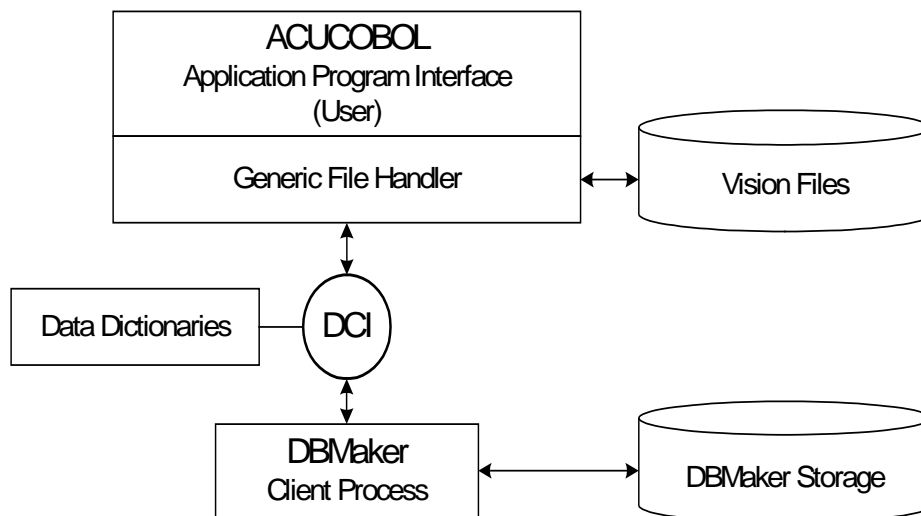


Figure 2-3 Data Flowchart

2.2 System Requirements

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

The README.TXT file located in the DCI directory lists the files that are shipped with the product.

The following platforms are supported by DCI:

- SCO OpenServer
- Sun Solaris x86
- Windows 9x/ME/NT/2000/XP
- Linux 2.2, 2.3
- AIX
- HP/UX
- FreeBSD 4

The following software must be installed for DCI to function:

- DBMaker version 4.2
- ACUCOBOL-GT runtime version 4.3 or higher
- A “C” compiler for the local machine (Visual C++™ Version 6.0 for a WINDOWS platform)

2.3 Setup Instructions

DBMaker version 4.2 must be installed and configured before configuring DCI. Refer to the Quick Start insert included with the DBMaker CD for instructions on installation of DBMaker.

Setup with Windows

The DCI files must be copied from the source directory on the DBMaker CD to a target directory before proceeding to setup DCI.

➤ To setup DCI:

1. Copy the DCI library `dmdcic.lib`, DBMaker library `dmapi42.lib` and the DCI library for Acu 5.1 or 5.2 from the DBMaker CD into the ACUCOBOL-GT installed directory.

For example

```
copy Drive:\DBMaker\4.2\lib\dmapi42.lib c:\acucobol\acugt\lib
copy CDROM:\DCI\WIN32\dmdcic.lib c:\acucobol\acugt\lib
```

To link DCI libraries with Acu 5.1 and previous versions:

```
copy CDROM:\DCI\WIN32\dmacu51.lib c:\acucobol\acugt\lib
```

To link DCI libraries with Acu 5.2 or newer versions:

```
copy CDROM:\DCI\WIN32\dmacu52.lib c:\acucobol\acugt\lib
```

2. Edit the ACUCOBOL runtime configuration file **filetbl.c**. It is in the directory that contains the ACUCOBOL-GT libraries, for example: `c:\acubl43\acugt\lib`. There are three entries you should modify:

- a) 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
```

- b) The original **filetbl.c** contains the entry:

```
extern DISPATCH_TBL v4_dispatch,...;
    extern DISPATCH_TBL ...;
```

Add a new entry as follows:

```
extern DISPATCH_TBL DBM_dispatch;
```

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

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

Add a new entry as follows:

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

3. Edit the ACUCOBOL runtime configuration file **sub85.c**. It is in the directory that contains the ACUCOBOL-GT libraries, for example: *c:\acucbl43\acugt\lib*.

The original **sub85.c** contains the entry:

```
struct PROCTABLE WNEAR LIBTABLE[] = {
    { "SYSTEM",    call_system },
```

Add a new entry as follows:

```
extern int DCI_GETENV();
extern int DCI_SETENV();
extern int DCI_DISCONNECT();
extern int DCI_GET_TABLE_NAME();
extern int DCI_SET_TABLE_CACHE();
extern int DCI_BLOB_ERROR();
extern int DCI_BLOB_PUT();
extern int DCI_BLOB_GET();
extern int DCI_GET_TABLE_SERIAL_VALUE();
extern int DCI_FREE_XFD();

struct PROCTABLE WNEAR LIBTABLE[] = {
{ "SYSTEM", call_system },
{ "DCI_GETENV", DCI_GETENV },
{ "DCI_SETENV", DCI_SETENV },
{ "DCI_DISCONNECT", DCI_DISCONNECT },
```

```
{ "DCI_GET_TABLE_NAME", DCI_GET_TABLE_NAME },
{ "DCI_SET_TABLE_CACHE", DCI_SET_TABLE_CACHE },
{ "DCI_BLOB_ERROR", DCI_BLOB_ERROR },
{ "DCI_BLOB_PUT", DCI_BLOB_PUT },
{ "DCI_BLOB_GET", DCI_BLOB_GET },
{ "DCI_GET_TABLE_SERIAL_VALUE", DCI_GET_TABLE_SERIAL_VALUE },
{ "DCI_FREE_XFD", DCI_FREE_XFD },
{ NULL, NULL }
};
```

4. Edit the ACUCOBOL file **direct.c** It is in the directory that contains the ACUCOBOL-GT libraries, for example: *c:\acucbl43\acugt\lib*.

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

```
struct EXTRNTABLE EXTDATA[] = {
    { NULL,          NULL }
};
```

Add a new entry as follows:

```
extern char *dci_where_constraint;
struct EXTRNTABLE EXTDATA[] = {
    { "DCI-WHERE-CONSTRAINT", (char *) &dci_where_constraint },
    { NULL,          NULL }
};
```

5. If using AcuGT < 6.0, open the file **wrun32.mak**. This is located in drive:\Acucorp\acucobol\acugt\lib. Add the **dmcdc.lib** and **dmapi42.lib** to **CLIENT_LIBS** or **LIBS** for ACUCOBOL 5.1 or ACUCOBOL 5.2, respectively. The files are located in the directory that contains the ACUCOBOL-GT libraries, if using ACUCOBOL 5.1 or previous versions install **dmacu51.lib** and if using ACUCOBOL 5.2 or newer versions install **dmacu52.lib**.

If using ACUCOBOL 5.1 search for **CLIENT_LIBS** in **wrun32.mak** and add the following library files:

```
CLIENT_LIBS=dmapi42.lib dmacu51.lib dmcdc.lib
```

If using ACUCOBOL 5.2 search for **LIBS** in **wrun32.mak** and add the following library files:

```
CLIENT_LIBS=\
    dmapi42.lib\
    dmacu52.lib\
```

```
dmdcic.lib\
```

6. If using ACUCOBOL 6.0 or 6.1, open the Visual C++ project named `wrun32.dsw` located in `lib\` directory of the AcuGT installation. Add the files `dmapi42.lib`, `dmacu52.lib`, `dmdcic.lib` to the project. Build the project to obtain the new `wrun32.dll` file.
7. If using ACUCOBOL 6.2, open the Visual .NET project named `wrundll.vcproj` located in the `lib\` directory of the AcuGT installation. Add the files `dmapi42.lib`, `dmacu52.lib`, and `dmdcic.lib` to the project. In the property, choose to use the MFC's common DLL. Build the project to obtain the new `wrun32.dll` file.
8. Open the command prompt and go to the directory that contains the ACUCOBOL-GT libraries, like: `c:\acucbl43\acugt\lib`.
9. If using AcuGT < 6.0, enter the command **`nmake -f wrun32.mak wrun32.exe`** at the command prompt to build the **`wrun32.exe`**.
If the `nmake` fails, execute the following to setup the VC++ environment:

```
vcvars32.bat <enter>
```
10. If using AcuGT < 6.0, enter the command **`nmake -f wrun32.mak wrun32.dll`** at the command prompt to build the **`wrun32.dll`**.
11. Copy the new **`wrun32.exe`** and **`wrun32.dll`** files to a directory mentioned in your execution path, for example `c:\acucbl43\acugt\bin`.
12. 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. Ensure it reports the version of the DBMaker interface.

Setup with UNIX

The DCI files must be copied from the source directory on the DBMaker CD (CDROM:\DCI\OS\) to a target directory before proceeding to setup DCI.

➤ To setup DCI:

1. Copy the DCI library `libdmdcic.a`, DBMaker library `libdmapic.a` and the DCI library for Acu 5.1 or 5.2 to the ACUCOBOL-GT installed directories:

For example: if a user wants to get DCI libraries for Linux, the user should issue the following command.

```
cp /home/dbmaker/4.2/bin/libdmapic.a /usr/acucobol/lib
```

```
cp /mnt/cdrom/dci/Linux2.x86/libdmdcic.a /usr/acucobol/lib
```

To link DCI libraries with Acu 5.1 and previous versions:

```
cp /mnt/cdrom/dci/Linux2.x86/libdmacu51.a /usr/acucobol/lib
```

To link DCI libraries with Acu 5.2 and newer versions:

```
cp /mnt/cdrom/dci/Linux2.x86/libdmacu52.a /usr/acucobol/lib
```

2. Edit the ACUCOBOL runtime configuration file **filetbl.c**. It is in the directory that contains the ACUCOBOL-GT libraries. There are three entries you should modify:

- a) 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
```

- b) 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 */
```

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

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

Add a new entry as follows:

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

3. Edit the ACUCOBOL runtime configuration file **sub85.c**. It is in the directory that contains the ACUCOBOL-GT libraries.

The original **sub85.c** contains the entry:

```
struct PROCTABLE WNEAR LIBTABLE[] = {
```



```
{ "SYSTEM",      call_system },
```

Add a new entry as follows:

```
extern int DCI_GETENV();
extern int DCI_SETENV();
extern int DCI_DISCONNECT();
extern int DCI_GET_TABLE_NAME();
extern int DCI_SET_TABLE_CACHE();
extern int DCI_BLOB_ERROR();
extern int DCI_BLOB_PUT();
extern int DCI_BLOB_GET();
extern int DCI_GET_TABLE_SERIAL_VALUE();
extern int DCI_FREE_XFD();

struct PROCTABLE WNEAR LIBTABLE[] = {
{ "SYSTEM", call_system },
{ "DCI_GETENV", DCI_GETENV },
{ "DCI_SETENV", DCI_SETENV },
{ "DCI_DISCONNECT", DCI_DISCONNECT },
{ "DCI_GET_TABLE_NAME", DCI_GET_TABLE_NAME },
{ "DCI_SET_TABLE_CACHE", DCI_SET_TABLE_CACHE },
{ "DCI_BLOB_ERROR", DCI_BLOB_ERROR },
{ "DCI_BLOB_PUT", DCI_BLOB_PUT },
{ "DCI_BLOB_GET", DCI_BLOB_GET },
{ "DCI_GET_TABLE_SERIAL_VALUE", DCI_GET_TABLE_SERIAL_VALUE },
{ "DCI_FREE_XFD", DCI_FREE_XFD },
{ NULL, NULL }
};
```

4. Edit the ACUCOBOL file **direct.c**. It is in the directory that contains the ACUCOBOL-GT libraries.

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

```
struct EXTRNTABLE EXTDATA[] = {
    { NULL,      NULL }
};
```

Add a new entry as follows:

```
extern char *dci_where_constraint;
struct EXTRNTABLE EXTDATA[] = {
```

```
{ "DCI-WHERE-CONSTRAINT",    (char *) &dc_i_where_constraint },
{ NULL,                      NULL }
};
```

5. Open the file Makefile. This is located in `/usr/acucobol/43/lib`. 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. If DBMaker has been installed in the directory `/DB/DBMaker` then the **Makefile** will contain the following string(s):

For ACUCOBOL 5.1 or previous versions:

```
FSI_LIBS=./libdmacu51.a libdmdcic.a ./libdmapic.a
```

For ACUCOBOL 5.2 or newer:

```
FSI_LIBS=./libdmacu52.a libdmdcic.a ./libdmapic.a
```

7. Next, ensure you are in the directory containing the ACUCOBOL-GT runtime system.

➤ Syntax 7a

At the UNIX prompt, type:

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

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

➤ Syntax 7b

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. Then to verify the link.

➤ Syntax 8a

Type:

```
./runcbl -vv
```

This will return version information on all of the products linked into your runtime system. Ensure it reports the version of DCI for DBMaker.

NOTE You may also link your own C routines with the runtime system.

9. 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

When you re-link the ACUCOBOL-GT runtime and try to execute it, you may receive an error message of this kind:

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

"Can't open shared library . . . "

That probably means that your operating system is using shared libraries but cannot locate them. This can happen even if the shared libraries exist in the current directory.

Each version of the UNIX operating system resolves this problem differently, so it is recommended that you look up your UNIX documentation on this problem. Some versions of UNIX require you to set an environment variable, which points to shared libraries on your system. For example, on an IBM RS/6000 running AIX 4.1, the environment variable LIBPATH have to indicate the directory where the shared libraries reside. On HP/UX, the environment variable is SHLIB_PATH. On UNIX SVR4, the environment variable is LD_LIBRARY_PATH. Please read the system documentation for your operating system to find the appropriate method to locate shared libraries.

Alternatively, you can link the shared libraries into the runtime with a static link to resolve this type of error. Each version of the C development system uses its own flag to accomplish this link. Please refer to the documentation for your C development system to find the correct flag for your environment.

2.4 Basic Configuration

Two configuration files must have parameters set for DCI to work. The first is **cbblconfig**, the ACUCOBOL runtime configuration file. The second is the **DCI_CONFIG** file that is located in a directory determined by an environment variable (see “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**

☞ Example

The following shows 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**

☞ Syntax

The following entry must be included in the configuration file.

```
DCI-DATABASE DBMaker_Test
```

NOTE *Refer to the section on “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 is given a username. The configuration variable DCI_LOGIN sets the username for all COBOL applications that use DCI. Initially this variable is 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.

➤ Syntax

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. 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.

➤ Syntax

If the database account is set to SYSADM, then the configuration file should appear as follows.

```
DCI_PASSWD
```

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.

➤ Syntax 1

To have data dictionaries stored in the directory */usr/DBMaker/Dictionaries*, it is necessary to include the following entry in the configuration file:

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

➤ **Syntax 2**

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
```

➤ **Syntax 3**

Using double-quotes in a WIN-32 environment can specify “embedded spaces”.

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

2.5 The Runsql Utility

DCI provides a utility program called runsql.acu. This program can access some of the standard SQL commands. It can be called from a COBOL program or executed from the command line. The SQL command may be up to 32767 characters in length and may be a variable or a quoted command string in the CALL statement

As a general rule, runsql.acu may be used to issue all SQL commands except those that perform data retrieval. The runsql.acu program cannot perform statements that return data such as the SELECT statement. This category of statements will return an error when passed to the runsql.acu program. The global variable return-code will be 0 when a command has completed successfully. If a command is not successful, the global variable return-code will contain an error code.

➤ Syntax 1

The following syntax is used to create a DBMaker view.

```
runcbl runsql.acu
```

➤ Example 1

The following is used to pause a program in order to accept an SQL command.

```
create table TEST (col1 char(10), col2 char(10))  
create view TESTW as select * from TEST
```

➤ Syntax 2

The following is used to call sql.acu from within a COBOL program.

```
call "runsql.acu" using sql-command
```

➤ Example 2

```
Call "runsql" using "create view TESTW as select * from TEST".
```

2.6 Invalid Data

DCI uses some methods to manage data that is valid in COBOL applications but invalid for the DBMaker RDBMS database. This section lists data types that cannot be accepted by the RDBMS and lists solutions that DCI implements to solve this problem.

COBOL VALUE	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
Zero	In DATE fields

Figure 2-4 Illegal COBOL Data

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.

Certain data types must be converted before DBMaker will accept them. DCI converts these values in the following ways:

- Illegal LOW-VALUES: stored as the lowest possible value (0 or - 99999...) or DCI_MIN_DATE default value.
- Illegal HIGH-VALUES: stored as the highest possible value (99999...) or DCI_MAX_DATE default value.
- Illegal SPACES: stored as zero (or DCI_MIN_DATE, in the case of a date field).
- Illegal DATE values: stored as DCI_INV_DATE default value.
- Illegal TIME: stored as DCI_INV_DATE 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.

If you want to change above conversion rules except for the key fields, you can use `DCI_NULL_ON_ILLEGAL_DATE` that convert to NULL illegal COBOL data.

2.7 Sample Application

A sample application program included with the DCI files demonstrates how DCI maps application data to a DBMaker database. This section can be used as a mini-lab to learn:

- How to setup 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, CBLCONFIG, INVD.XFD, DCI.CFG, 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).

NOTE *To compile the sample application, users should already have installed ACUCOBOL 4.3 or above.*

➡ To run the application:

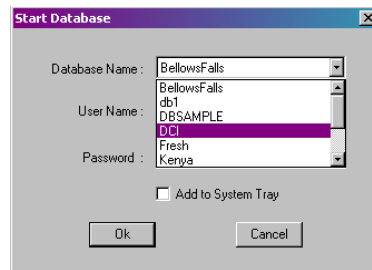
1. Setup a database in DBMaker to accept data from DCI. As an example for this procedure, we created the database DCI using JServer Manager. All default settings were used for the database; specifically SYSADM was used for the default login name with no password. For information on creating and setting 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.CFG file with any text editor. Set the configuration variables to appropriate values. Refer to section 2.4 “Basic Configuration” for information on configuration file values.

➡ Example

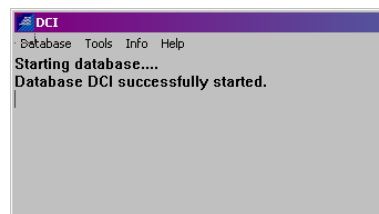
DCI_DATABASE	DCI
--------------	-----

```
DCI_LOGIN      SYSTEM
DCI_PASSWD
//DCI_LOGFILE
DCI_STORAGE_CONVENTION Dca
//DCI_XFDPATH C:\DCI
```

3. Run the DBMaker Server program (dmserver.exe). 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 will 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 Guide*.



5. Open the command prompt and go to the `..\DCI` directory.
6. Define the DCI_CONFIG at the command prompt by entering the following.

➤ **Syntax 6a**

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

7. Run the COBOL program INVD.ACU using WRUN32.

➤ **Syntax 7a**

At the command prompt in the same directory, enter the following line:

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

8. The file CBLCONFIG contains the command line DEFAULT-HOST DCI and is used to set the default file system. For more information refer to Chapter 5, "Compiler and Runtime Options".
9. The window of the COBOL application INVD.ACUI will open as shown below, allowing you to enter values into the fields.

NOTE For instructions on adding records, refer to "Adding Records" below.

➤ **To compile the sample application from the source code:**

1. Follow steps 1 through 6 in "To run the application", 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`.

NOTE ACUCOBOL 4.3 users should copy the above definition files from the `\Acucbl43\AcuGT\sample\`

3. At the command prompt go to the `..\DCI` directory.

➤ **Syntax 3a**

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 and data dictionary file INVD.XFD. To run the object file follow steps 6 and 7 in “To run the application”, 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 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 the **Add** button. The values entered into the fields will now be saved in the DBMaker database specified by the DCI.CFG variable, DCI-DATABASE.

➞ Example

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-INVLSSTKNO	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.

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.

The screenshot shows a window titled "Screen" with a menu bar containing "First", "review", "Next", "Last", a dropdown menu, "Refresh", "Add", "Update", "Delete", "Clear", and "Exit". Below the menu bar, there are several fields with labels 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

Figure 2-5 INVD-INVLNO Key Field Sample Entry

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*, 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.

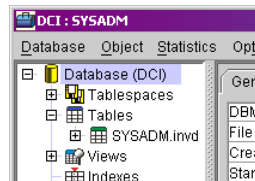


Figure 2-6 INVD Application Tables Tree Node

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.

Name	Type	Precision	Scale	Nullable	Primary	Default Value	Constraints
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	✓			

Figure 2-7 SYSADM.invd Table Schema

Selecting the **Edit Data** tab will allow you to view the values of each field.

INVD_INVLNO	INVD_INVLSTK	INVD_INVLDESC	INVD_INVLQTY	INVD_INVLFREE	INVD_MVTCODE	INVD_SUBTOTAL
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	

Figure 2-8 SYSADM.invd Edit Data Tab Field Values

3 Data Dictionaries

Data Dictionaries describe how extended file descriptor (.XFD) files are created and accessed. DCI avoids using SQL function calls embedded in COBOL code by using a special feature of ACUCOBOL-GT. When a COBOL application is compiled using the “-Fx” option data dictionaries are generated. These are known as “extended file descriptors” (XFD files), which are based on COBOL file descriptors. 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 *Refer to the ACUCOBOL-GT User's Guide (chapter 5.3) for more detailed information and rules concerning the creation of XFDs.*

3.1 Assigning Table Names

Database tables correspond to file descriptors in the FILE CONTROL section of the COBOL application. The database tables must have unique names under 32 bytes in length (32 ASCII characters).

It is possible for the DBMaker table to have more columns than the COBOL program's corresponding file descriptor. It is also possible to have different orders columns than the COBOL program's corresponding file descriptor.

The number of columns in the database table does not have to match up with the number of fields in the COBOL program that is accessing the table. The DBMaker table can have more columns than the COBOL program references; however, the COBOL program may not have more fields than the DBMaker table. Ensure that these "extra" columns are set correctly when new rows are added to the table.

ACUCOBOL generates XFD file names by default from the FILE CONTROL section. 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 Directive in chapter 4). 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. If the ASSIGN phrase refers to a device and is generic (such as ASSIGN TO "DISK"), then the compiler uses the SELECT name to generate the XFD file name.

File names and usernames are case-insensitive. All file descriptors containing uppercase characters will be converted to lowercase. Users must be aware of this if using a case sensitive operating system.

➤ Example 1

If the FILE CONTROL section contains the following line of text:

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

➤ Example 2

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 lowercase. 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.

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

➤ Example 3

If the file has a file extension, DCI replaces “.” characters with “_”. DCI will open a DBMaker table named “username.customer_dat”.

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

➤ Example 4

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 would 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

Figure 3-1 Sample DCI_USEDIR_LEVEL to “2” Table

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.

COBOL CODE	RESULTING FILE NAME	RESULTING TABLE NAME
ASSIGN TO "usr/hr/employees.dat"	employees_dat.xfd	employees_dat
SELECT DATAFILE, ASSIGN TO DISK	datafile.xfd	datafile
ASSIGN TO "-D SY\$LIB:EMP"	emp.xfd	emp
ASSIGN TO FILENAME	(user specified)	(user specified)

Figure 3-2 Example Table Names Formed From Different COBOL Statements

➤ Example

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.

```
05 DATE-PURCHASED.
   10 YYYY          PIC 9(04).
   10 MM            PIC 9(02).
   10 DD            PIC 9(02).
05 PAY-METHOD     PIC X(05).
```

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 32 characters and converts it to lowercase depending of DCI_CASE value.

3.2 Mapping Columns and Records

The table that is created is based on the largest record in the COBOL file. It contains all of 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.

➞ Example 1

The following 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(06).
    05 EMP-NAME        PIC X(17).
    05 EMP-PHONE       PIC X(10).

WORKING-STORAGE SECTION.
01  HR-NUMBER-FIELD    PIC 9(05).

PROCEDURE DIVISION.
PROGRAM-BEGIN.
    OPEN I-O HR-FILE.
    PERFORM GET-NEW-EMPLOYEE-ID.
    PERFORM ADD-RECORDS UNTIL EMP-ID = ZEROS.
    CLOSE HR-FLE.
PROGRAM-DONE.
    STOP RUN.
GET-NEW-EMPLOYEE-ID.
  
```

```
PERFORM INIT-EMPLOYEE-RECORD.  
PERFORM ENTER-EMPLOYEE-ID.  
INIT-EMPLOYEE-ID.  
    MOVE SPACES TO EMPLOYEE-RECORD.  
    MOVE ZEROS TO EMP-ID.  
ENTER-EMPLOYEE-ID.  
    DISPLAY "ENTER EMPLOYEE ID NUMBER (1-99999),"  
    DISPLAY "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.
```

➞ Example 2

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 file 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.

➞ Example 3

The database will create a distinct column for each field in the file descriptor. The table name will be HR-FILE in accordance with the SELECT statement in the FILE-CONTROL section. The database records in the example would therefore have the following structure:

EMP_ID (INT(6))	EMP_NAME (CHAR(17))	EMP_PHONE (DEC(10))
51100	LAVERNE HENDERSON	2221212999
52231	MATTHEW LEWIS	2225551212

Figure 3-3 Table EMPLOYEE-RECORD

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 “mapping” 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. 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.

➔ Example

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).
----	-----------	------------

Long Field Names

DBMaker allows for table names up to 32 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 32 characters. For example, if the field name were Employee-statistics-01 it would be truncated to form the table name Employee_statistic_01. It is important to ensure 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 Using 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.

➡ Example 1

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.

EMP_ID (INT(6))	EMP_NAME (CHAR(17))	EMP_PHONE (DEC(10))
-----------------	---------------------	---------------------

Figure 3-4 Preceding Example Table

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.

➤ Example 2

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
```

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 is 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.

➤ **Example 3**

An SQL select on the first record of all columns in EMP_NAME would display the following:

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

➤ **Example 4**

An SQL select on the second record of all columns in EMP_NAME would display the following:

```
500000, 0000010400
```

3.4 Using XFD File Defaults

In many cases directives can be used to override the default behavior of DCI. Refer to XFD Directives for more information.

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

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

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 original field. By default, the data dictionary uses the field definition of the subordinate field to define the column data type.

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.

➞ Example 1

Given the following part of a file descriptor:

```
03 EMPLOYEE-RECORD OCCURS 20 TIMES.
    05 CUST-ID                               PIC 9(5).
```

➞ Example 2

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
```

3.5 Mapping Multiple Files

It is possible at runtime to use a single XFD file for multiple files with different names. If the record definitions of the files are the same then it is unnecessary to create a separate XFD for each file.

The runtime configuration variable `DCI_MAPPING` determines which files are mapped to an XFD. Below is a description of how it works.

Suppose the COBOL application has a `SELECT` with a variable `ASSIGN` name, such as `EMPLOYEE-RECORD`. This variable assumes different values (such as `EMP0001` and `EMP0002`) during program execution. In order to provide a base name for the XFD, use the `FILE` directive (see ((XFD DATE, USE GROUP))).

➞ Example

If “EMP” is the base, then the compiler will generate an XFD named “Emp.xfd”. The asterisk (“*”) in the example is a wildcard character that replaces any number of characters in the file name. The file extension “.xfd” is not included in the map. This statement would cause the XFD “emp.xfd” to be used for all files with names that begin with “EMP”. Add the following entry in the runtime configuration file to ensure that all employee files, each having a unique but related name, use the same XFD:

```
DCI_MAPPING EMP* = EMP
```

The `DCI_MAPPING` variable is read during the open file stage. The “*” and “?” wildcard characters can be used within the pattern:

* matches any number of characters

? matches a single occurrence of any character

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

➤ Syntax

Where *<pattern>* consists of any valid filename characters and may include “*” or “?”. The DCI_MAPPING variable has the following syntax:

```
DCI_MAPPING [<pattern> = base-xfd-name],
```

3.6 Mapping to Multiple Databases

It is possible to reference tables in different databases with DCI_DB_MAP by specifying different files or COBOL file-prefix links to the DBMS. This scenario is illustrated through the following example.

➞ Example

To reference table **idx1** in the databases DBSAMPLE4 (as default), DBCED, and DBMULTI, add the following settings in the DCI_CONFIG configuration file.

```
DCI_DB_MAP      /usr/CED=DBCED
DCI_DB_MAP      /usr/MULTI=DBMULTI
```

To create the **idx1** table in these databases by specifying different files:

```
...
    INPUT-OUTPUT SECTION.
    FILE-CONTROL.

        SELECT IDX-1-FILE
        ASSIGN TO DISK "/usr/CED/IDX1"
        ORGANIZATION IS INDEXED
        ACCESS IS DYNAMIC
        RECORD KEY IS IDX-1-KEY.

        SELECT IDX-2-FILE
        ASSIGN TO DISK "/usr/MULTI/IDX1"
        ORGANIZATION IS INDEXED
        ACCESS IS DYNAMIC
        RECORD KEY IS IDX-2-KEY.

        SELECT IDX-3-FILE
        ASSIGN TO DISK "IDX1"
        ORGANIZATION IS INDEXED
        ACCESS IS DYNAMIC
        RECORD KEY IS IDX-3-KEY.

DATA DIVISION.
FILE SECTION.
FD  IDX-1-FILE.
```



```

01  IDX-1-RECORD.
    03  IDX-1-KEY                                PIC X(10).
    03  IDX-1-ALT-KEY.
        05  IDX-1-ALT-KEY-A                    PIC X(30).
        05  IDX-1-ALT-KEY-B                    PIC X(10).
    03  IDX-1-BODY                                PIC X(50).

FD  IDX-2-FILE.
01  IDX-2-RECORD.
    03  IDX-2-KEY                                PIC X(10).
    03  IDX-2-ALT-KEY.
        05  IDX-2-ALT-KEY-A                    PIC X(30).
        05  IDX-2-ALT-KEY-B                    PIC X(10).
    03  IDX-2-BODY                                PIC X(50).

FD  IDX-3-FILE.
01  IDX-3-RECORD.
    03  IDX-3-KEY                                PIC X(10).
    03  IDX-3-ALT-KEY.
        05  IDX-3-ALT-KEY-A                    PIC X(30).
        05  IDX-3-ALT-KEY-B                    PIC X(10).
    03  IDX-3-BODY                                PIC X(50).

WORKING-STORAGE SECTION.

PROCEDURE DIVISION.
LEVEL-1 SECTION.
MAIN-LOGIC.
    set environment "default-host" to "dci"

*   make IDX1 table on DBCED

    OPEN OUTPUT IDX-1-FILE
    MOVE "IDX IN DBCED" TO IDX-1-BODY
    MOVE "A" TO IDX-1-KEY
    WRITE IDX-1-RECORD
    MOVE "B" TO IDX-1-KEY
    WRITE IDX-1-RECORD
    MOVE "C" TO IDX-1-KEY

```

```

        WRITE IDX-1-RECORD
        CLOSE IDX-1-FILE

*   make IDX1 table on DBMULTI
    OPEN INPUT IDX-1-FILE
    OPEN OUTPUT IDX-2-FILE
    PERFORM UNTIL 1 = 2
        READ IDX-1-FILE NEXT AT END EXIT PERFORM END-READ
        MOVE IDX-1-RECORD TO IDX-2-RECORD
        MOVE "IDX IN DBMULTI" TO IDX-2-BODY
        WRITE IDX-2-RECORD
    END-PERFORM
    CLOSE IDX-1-FILE IDX-2-FILE

*   make IDX1 table on DBSAMPLE4
    OPEN INPUT IDX-1-FILE
    OPEN OUTPUT IDX-3-FILE
    PERFORM UNTIL 1 = 2
        READ IDX-1-FILE NEXT AT END EXIT PERFORM END-READ
        MOVE IDX-1-RECORD TO IDX-3-RECORD
        MOVE "IDX IN DBSAMPLE4" TO IDX-3-BODY
        WRITE IDX-3-RECORD
    END-PERFORM

    CLOSE IDX-1-FILE IDX-3-FILE

```

To read table **idx-1** in these databases by file-prefix:

```

.....
INPUT-OUTPUT SECTION.
FILE-CONTROL.

    SELECT IDX-1-FILE
    ASSIGN TO DISK "IDX1"
    ORGANIZATION IS INDEXED
    ACCESS IS DYNAMIC
    RECORD KEY IS IDX-1-KEY.

DATA DIVISION.
FILE SECTION.

```

```
FD  IDX-1-FILE.
01  IDX-1-RECORD.
    03  IDX-1-KEY                      PIC X(10).
    03  IDX-1-ALT-KEY.
        05  IDX-1-ALT-KEY-A          PIC X(30).
        05  IDX-1-ALT-KEY-B          PIC X(10).
    03  IDX-1-BODY                      PIC X(50).

WORKING-STORAGE SECTION.

PROCEDURE DIVISION.
LEVEL-1 SECTION.
MAIN-LOGIC.
    set environment "default-host" to "dci"

    set environment "file-prefix" to "/usr/MULTI:/usr/CED".
    OPEN INPUT IDX-1-FILE
    READ IDX-1-FILE NEXT
    DISPLAY IDX-1-BODY
    ACCEPT OMITTED
    CLOSE IDX-1-FILE

    set environment "file-prefix" to "/usr/CED:/usr/MULTI".
    OPEN INPUT IDX-1-FILE
    READ IDX-1-FILE NEXT
    DISPLAY IDX-1-BODY
    ACCEPT OMITTED
    CLOSE IDX-1-FILE

    set environment "file-prefix" to " ./usr/CED:/usr/MULTI".
    OPEN INPUT IDX-1-FILE
    READ IDX-1-FILE NEXT
    DISPLAY IDX-1-BODY
    ACCEPT OMITTED
    CLOSE IDX-1-FILE
```

3.7 Using Triggers

COBOL Triggers are very useful and powerful features of DCI. COBOL triggers can be used to automatically execute predefined COBOL program in response to specific I/O events, regardless of which user or application program generated them.

COBOL triggers can be used to:

- Implement business rules.
- Create an audit trail for COBOL activities.
- Derive additional values from existing data.
- Replicate data across multiple files.
- Perform security authorization procedures.
- Control data integrity.
- Define unconventional integrity constraints.

Use the following XFD directives to define a COBOL trigger in order to specify the COBOL program name to be called when an I/O event occurs.

➞ Syntax

```
$XFD DCI COMMENT COBTRIGGER "cobolpgmname"
```

➞ Example 1

The “cobolpgmname” is case-sensitive and looks in the CODE-PREFIX directory or current running directory. The I/O events may be READ (any), WRITE, REWRITE, DELETE, and OPEN. The COBOL trigger performs BEFORE and AFTER I/O events except for OPEN that performs BEFORE I/O events.

```
$xfd dci comment cobtrigger "cobtrig"
```

➞ Example 2

The “cobolpgmname” must following the LINKAGE SECTION rule :

```
LINKAGE SECTION.
```

```
01    op-code    PIC x.
88    read-after    value "R".
88    read-before    value "r".
```

```

88  write-after      value "W".
88  write-before    value "w".
88  rewrite-after    value "U".
88  rewrite-before   value "u".
88  delete-after     value "D".
88  delete-before    value "d".
88  open-before      value "O".

01  record-image     PIC x(32767).

01  rc-error         PIC 99.

```

➡ Example 3

Op-code is valued from DCI based on I/O events. The *record-image* contains the COBOL record value before/after the I/O events. The *rc-error* could be used to force the COBOL I/O events error using the following values:

```

88  F-IN-ERROR              VALUES 1 THRU 99.
88  E-SYS-ERR               VALUE 1.
88  E-PARAM-ERR             VALUE 2.
88  E-TOO-MANY-FILES        VALUE 3.
88  E-MODE-CLASH            VALUE 4.
88  E-REC-LOCKED            VALUE 5.
88  E-BROKEN                VALUE 6.
88  E-DUPLICATE             VALUE 7.
88  E-NOT-FOUND             VALUE 8.
88  E-UNDEF-RECORD          VALUE 9.
88  E-DISK-FULL             VALUE 10.
88  E-FILE-LOCKED          VALUE 11.
88  E-REC-CHANGED           VALUE 12.
88  E-MISMATCH              VALUE 13.
88  E-NO-MEMORY             VALUE 14.
88  E-MISSING-FILE          VALUE 15.
88  E-PERMISSION            VALUE 16.
88  E-NO-SUPPORT            VALUE 17.
88  E-NO-LOCKS              VALUE 18.
88  E-INTERFACE             VALUE 19.

```

3.8 Using Views

DCI allows the use of DBMaker views instead of a table. In this case DCI users must manually create a view and be aware of the following limitations:

- Users can open a view and do all DML operations when the view is a single table view and the projection column on the original table is without an expression, aggregate or UDF.
- For other kinds of views, user can open the view as an OPEN INPUT and perform a READ operation only.

🔗 Example 1

The example below shows how to create the view in the COBOL program and open it. It assumes that there are 2 tables named **t2** and **t3** created as follows:

```
create table t2 ( c1 char(30), c2 int);
create table t3(c1 int);
```

and that such tables are filled with some data.

```
identification division.

file-control.
    select miofile assign to ws-nomefile
        organization indexed
        access mode dynamic
        record key rec
        .
data division.
file section.
$XFD FILE=miofile
fd miofile.
01 rec.
    03 c1 pic x(30).
    03 c2 pic 9(9).

working-storage section.
01 ws-nomefile pic x(30).
01 sql-command pic x(1000).
```

```
procedure division.  
main.  
    set environment "default_host" to "dci"  
    display "Enter the name of the view to create:" no  
    accept ws-nomefile  
  
    inspect ws-nomefile replacing trailing spaces  
        by low-value  
  
    string "create view " delimited by size  
        ws-nomefile delimited by low-value  
        " as (select c1, c2 from t2 where c2 in (select max(c1)  
-        " from t3));" delimited by size  
        x"00" delimited by size  
        into sql-command  
  
    display sql-command  
    accept omitted  
  
    call "i$io" using 15, "dci", sql-command  
    if return-code not = 0  
        display "Errore : " return-code  
        accept omitted  
        stop run  
    end-if  
  
    string "commit;" delimited by size  
        x"00" delimited by size  
        into sql-command  
    call "i$io" using 15, "dci", sql-command  
    if return-code not = 0  
        display "Errore : " return-code  
        accept omitted  
        stop run  
    end-if  
    open input miofile  
    perform until 1=2  
        read miofile next  
        at end exit perform
```

```
end-read  
display rec  
end-perform  
close miofile  
  
exit program
```


3.9 Using Synonyms

DCI allows the use of DBMaker synonyms instead of a table or view. Users can create the synonym on a table, view or remote database's table or view. If the synonym for the view is not a single table view, the user can only OPEN INPUT with that synonym.

3.10 Open Tables in Remote Databases

Users can access the remote database's table or view by adding a special token “@” in the COBOL SELECT statement. For example:

```
SELECT tbl ASSIGN TO RANDOM, "lnk1@tbl"
```

The user must set DD_DDBMD=1 in the dmconfig.ini and create a remote database link if the user wants to use a different user name and password.

🔗 Example

You want to connect to database **dci_db1** and access a table in database **dci_db2**.

1. Set DD_DDBMD=1 in dmconfig.ini.
2. Then create the table in dci_db2

NOTE Use the dmSQL tool to create the tables in the dci_db2 database

3. Use a COBOL program to connect to dci_db1 and then open the table in dci_db2.

```
dmconfig.ini
[DCI_DB1]
DB_SVADR = 127.0.0.1
DB_PTNUM = 22999
DD_DDBMD = 1
```

```
[DCI_DB2]
DB_SVADR = 127.0.0.1
DB_PTNUM = 23000
DD_DDBMD = 1
```

```
Use dmSQL tool to create the table
connect to DCI_DB2 SYSADM;
create table tbl (c1 int not null, c2 int, c3 char(10), primary key c1);
commit;
disconnect;
```

```
COBOL program
identification division.
program-id.RemoteTable.
```

```

date-written.
remarks.
environment division.
input-output section.
file-control.
        SELECT tbl ASSIGN TO RANDOM, "dci_db2@tbl"
        ORGANIZATION IS INDEXED
        ACCESS IS DYNAMIC
        FILE STATUS IS I-O-STATUS
        RECORD KEY IS C1.

data division.
file section.
FD  tbl.
01  tbl-record.
        03 C1          PIC 9(8) COMP-5.
        03 C2          PIC 9(8) COMP-5.
        03 C3          PIC X(10).

working-storage section.
77  I-O-STATUS pic xx.

procedure division.
main.
        set environment "default_host" to "dci"
        call "DCI_SETENV" using "DCI_DATABASE" "DCI_DB1"
        call "DCI_SETENV" using "DCI_LOGIN" "SYSADM"

        open i-o tbl
        move 100 TO C1.
        move 200 TO C2.
        move "AAAAAAAAAA" TO C3.
        write tbl-record.
        initialize tbl-record.
        read tbl next.
        display C1, C2, " ", C3.
        close tbl.
        accept omitted.
        stop run.

```

3.11 Using DCI_WHERE_CONSTRAINT

DCI_WHERE_CONSTRAINT is used to specify an additional WHERE condition for a succeeding START operation. To be compatible with Acu4gl, DCI also supports the 4gl_where_constraint.

➡ Example:

If you want to query city names that start with **A**, add the following to your code:

```
WORKING-STORAGE SECTION.
01 dci_where_constraint pic x(4095) is external.
...

PROCEDURE DIVISION.
...

* to specify dci_where_constraint
move low-values to dci_where_constraint
  open i-o idx-1-file
  move "city_name = 'a%'" to dci_where_constraint
  inspect dci_where_constraint replacing trailing spaces by low-values.

move spaces to idx-1-key
start idx-1-file key is not less idx-1-key
....
```

4 XFD Directives

Directives are comments placed in COBOL file descriptors that alter how the database table is built. Directives can be used to change the way data is defined in the database and to assign names to database fields. The directives can also assign names to .XFD files, assign data to binary large object (BLOB) fields, or add comments.

4.1 Using Directive Syntax

Each directive is placed on a line by itself, immediately before the related line of COBOL code. All directives have the prefix \$XFD; a \$ symbol in the 7th column followed immediately by XFD.

➞ Syntax 1

The following command provides a unique database name for an undefined COBOL variable. The directive is issued above the line it is intended to affect; in this case the second instance of the COBOL defined variable *qty*.

```
. . .
  03 QTY          PIC 9(03).
  01 CAP.
$XFD NAME=CAPQTY
  03 QTY          PIC 9(03).
```

➞ Syntax 2

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

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

➞ Syntax 3

More than one directive may be combined together. Directives can be on the same line, preceded by the prefix \$XFD and separated by a space or comma.

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

➞ Syntax 4

Alternatively, the following can be used.

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

4.2 Using XFD Directives

Directives are used when a COBOL file descriptor is mapped to a database field. The \$XFD prefix indicates to the compiler that the proceeding command is used during the generation of the data dictionary.

\$XFD ALPHA Directive

In order to store non-numeric data (like, 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.

➔ Syntax 1

```
$XFD ALPHA
```

➔ Syntax 2

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

Moving a non-numeric value such as “A234” to the key without using the \$XFD ALPHA directive would

➔ Example 1

Let’s establish that the KEY IS code-key has been specified and we have the following record definition. CODE-NUM is a numeric value and is the key field here, since group items are disregarded in the database.

```
01 EMPLOYEE-RECORD.
   05 EMP-KEY.
      10 EMP-NUM          PIC 9(5).
```

➔ Example 2

Using the \$XFD ALPHA directive will change a non-numeric value such as “A234” so that the record will not be rejected by the database, since “A234” is an alphanumeric value and CODE-NUM is a numeric value.

```
01 EMPLOYEE-RECORD.
   05 EMP-KEY.
   $XFD ALPHA
```

```
10 EMP-NUM          PIC 9(5)
```

➞ Example 3

Now, the following operation can be used without worrying about any rejection .

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

\$XFD BINARY Directive

In order to allow for the data 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.

➞ Syntax 1

```
$XFD BINARY
```

➞ Syntax 2

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

➞ Example

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

```
01 EMPLOYEE-RECORD.
  05 EMP-KEY.
    10 EMP-TYPE      PIC X.
  $(( XFD BINARY ))
    10 EMP-NUM       PIC 9(05).
    10 EMP-SUFFIX    PIC X(03).
```

\$XFD COMMENT DCI SERIAL n Directive

This directive is used to define a serial data field and an optional starting number “n”. To trigger DBMaker to generate a serial number, insert a record and supply 0 value for the serial field. If you insert a new row and supply an integer value instead of a 0 value, *DBMaker* will not generate a serial number. If the supplied

integer value is greater than the last serial number generated, *DBMaker* will reset the sequence of generated serial numbers to start with the supplied integer value.

➤ **Syntax 1**

```
$XFD COMMENT DCI SERIAL 1000
```

➤ **Syntax 2**

```
*(( XFD COMMENT DCI SERIAL 1000 ))
```

➤ **Example**

```
01 EMPLOYEE-RECORD.  
    05 EMP-KEY.  
        10 EMP-TYPE      PIC X.  
$(( XFD COMMENT DCI SERIAL 250 ))  
        10 EMP-COUNT     PIC 9(05).
```

\$XFD COMMENT DCI COBTRIGGER Directive

This directive lets you to define a COBOL program as trigger of I/O events like READ WRITE REWRITE or DELETE. This defined COBOL program is automatic called before and after every I/O events.

➤ **Syntax 1**

```
$XFD COMMENT DCI COBTRIGGER "cblprogramname"
```

➤ **Syntax 2**

```
*(( XFD COMMENT DCI COBTRIGGER "cblprogramname" ))
```

\$XFD COMMENT Directive

This directive is used to include comments in an XFD file. In this way, information can be embedded 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 DCI interfaces. Each comment will be recognizable in the XFD file as having the “#”symbol in column 1.

➤ **Syntax 1**

```
$XFD COMMENT text
```

☞ Syntax 2

```
*(( 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.

☞ Syntax 1

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

☞ Syntax 2

```
*((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.

CHARACTER	DESCRIPTION
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

Figure 4-1 date-format-string Characters

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

The definition of Julian dates varies, so the DATE directive allows for a flexible representation of Julian dates. Many sources define the Julian day as the day of the year, with January 1st being 001, January 2nd being 002, etc. To use this definition for Julian day, simply use FEE (day of year) in the date formats. Other references define the Julian day as the number of days since a specific base date. This definition is represented in the DATE directive by the letter J (for example, a six-digit date field would be preceded with the directive \$XFD DATE=JJJJJ). 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 `DCI_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.

➞ Example 1

```
$XFD DATE
  05 DATE-PURCHASED      PIC 9(08).
  05 PAY-METHOD         PIC X(05).
```

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

➞ Example 2

```
$(( XFD DATE, USE GROUP ))
  05 DATE-PURCHASED.
    10 YYYY          PIC 9(04).
    10 MM            PIC 9(02).
    10 DD            PIC 9(02).
  05 PAY-METHOD     PIC X(05).
```

\$XFD FILE Directive

The FILE directive names the data dictionary with the file extension .XFD. This directive is required when creating 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.

➤ Syntax 1

```
$XFD FILE=filename
```

➤ Syntax 2

```
*(( XFD FILE=filename ))
```

➤ Example

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

```
ENVIRONMENT DIVISION.
FILE-CONTROL.
SELECT FILENAME ASSIGN TO VARIABLE-OF-WORKING.
. . .
DATA DIVISION.
FILE SECTION.
$XFD FILE=CUSTOMER
FD FILENAME
. . .
```

\$XFD NAME Directive

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

➤ Syntax 1

```
$XFD NAME=columnname
```

➤ Syntax 2

```
*(( XFD NAME=columnname ))
```

➤ **Example**

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

```
$XFD NAME=customercode
      05 cus-cod          PIC 9(05).
```

\$XFD NUMERIC Directive

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

➤ **Syntax 1**

```
$XFD NUMERIC
```

➤ **Syntax 2**

```
*(( XFD NUMERIC ))
```

➤ **Example**

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

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

\$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), where n=1-max column length). The directive may be combined with other directives if the data is stored as a different type (BINARY, DATE, NUMERIC). Combining fields into groups improves processing speed on the database, so effort is made to determine which fields can be combined.

➤ **Syntax 1**

```
$XFD USE GROUP
```

➤ **Syntax 2**

```
*(( XFD USE GROUP ))
```

➤ Example 1

By adding the USE GROUP directive, the data is stored as a single numeric field where the column name is *code-key*.

```
01 CODE-RECORD.
$XFD USE GROUP
    05 CODE-KEY.
        10 AREA-CODE-NUM    PIC 9(03).
        10 CODE-NUM        PIC 9(07).
```

➤ Example 2

The USE GROUP directive can be combined with other directives. The fields are mapped into a single DATE type data column in the database.

```
$(( XFD DATE, USE GROUP ))
    05 DATE-PURCHASED.
        10 YYYY            PIC 9(04).
        10 MM              PIC 9(02).
        10 DD              PIC 9(02).
```

\$XFD VAR-LENGTH Directive

VAR-LENGTH 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 COBOL **conversion**). 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.

➤ Syntax 1

```
$XFD USE VAR-LENGTH
```

➤ Syntax 2

```
*(( XFD USE VAR-LENGTH ))
```

➤ Example

```
$XFD USE VAR-LENGTH
    05 LARGE-FIELD    PIC X(10000).
```

\$XFD WHEN Directive for File Names

The WHEN directive is used to build certain columns in DBMaker 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.

☞ Syntax 1

(Equal to)

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

☞ Syntax 2

(Less than or equal to)

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

☞ Syntax 3

(Less than)

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

☞ Syntax 4

(Greater than or equal to)

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

☞ Syntax 5

(Greater than)

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

☞ Syntax 6

(Not equal to)


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

➞ Syntax 7

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.

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

➞ Syntax 8

Value is an explicit data value used in quotes, and field is a previously defined COBOL field.

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

➞ Example

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

```
05 AR-CODE-TYPE          PIC X.
$XFD WHEN AR-CODE-TYPE="S"
05 SHIP-CODE-RECORD      PIC X(04).
$XFD WHEN AR-CODE-TYPE="B"
05 BACKORDER-CODE-RECORD REDEFINES SHIP-CODE-RECORD.
$XFD WHEN AR-CODE-TYPE=OTHER
05 OBSOLETE-CODE-RECORD  REFEFINES SHIP-CODE-RECORD.
```

TABLENAMENAME OPTION

The WHEN directive has the TABLENAMENAME option to change the table name according to the value of the WHEN directive during runtime.

When using the TABLENAMENAME option in a WHEN statement, be aware of the DCI_DEFAULT_RULES and filename_RULES DCI configuration variables.

➞ Example 1

A COBOL FD structure using the “When” directive with two table names.

```
FILE SECTION.
$XFD FILE=INV
```

```

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).

```

➡ Example 2

The DCI interface makes two tables named “inv-top” and “inv-details” based on the value of the inv-type fields in example 1. DCI 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 preceding 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 through record types, you must set `DCI_DEFAULT_RULES = POST` or `= COBOL`. Alternately, if you plan to use sequential reading inside record types you must set `DCI_DEFAULT_RULES=BEFORE` or `= DBMS`.

There are advantages and disadvantages to using this rule. To have a 100% COBOL ANSI reading behavior, you should use the “POST” or “COBOL” 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” or “DBMS” method, the involved table is opened when the \$WHEN condition matches at the read record level.

☞ Example 3

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

```
OPEN INPUT INVOICE.
* to see the customer 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” or “COBOL”, the “open input” opens both tables and “read next”, reads thru different tables.

☞ Example 4

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

If the method is “BEFORE” or “DBMS” the code is 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
```

\$XFD COMMENT DCI SPLIT

The DCI SPLIT directive is used to define one or more table splitting points starting where the DCI interface makes a new DBMS table.

➡ Example 1

A COBOL FD structure using DCI SPLIT directive.

In this example three DBMaker tables named INVOICE, INVOICE_A, and INVOICE_B are created with fields between the split points.

```
FILE SECTION.
FD  INVOICE.
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 DCI SPLIT
    03  INV-KEY-D.
        05  INV-TYPE-D         PIC X.
        05  INV-NUMBER-D       PIC 9(5).
        05  INV-ID-B          PIC 999.
$XFD DCI SPLIT
    03  INV-ARTICLES          PIC X(30).
    03  INV-QTA               PIC 9(5).
    03  INV-PRICE             PIC 9(17).
```

5 Compiler and Runtime Options

This section describes configuration settings for ACUCOBOL-GT used to specify what file system to use.

5.1 Using ACUCOBOL-GT Default File System

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

The DEFAULT-HOST setting tells ACUCOBOL which file 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 file system for a specific file. The name of the file should replace *filename* in the setting.

The following variables in the ACUCOBOL-GT configuration file allow the file system of choice to be used.

➤ Syntax 1

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

➤ Syntax 2

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

5.2 Using DCI Default File 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 file system. If no file system is specified, the Vision file system will be used by default.

➤ Syntax 1

In this case, all new files will be DBMaker files, unless the new files have been designated to a different file system.

```
DEFAULT-HOST DCI
```

➤ Syntax 2

In order to establish that all new files, unless otherwise specified, will be Vision files, use the following.

```
DEFAULT-HOST VISION
```

5.3 Using Multiple File Systems

Filename-HOST is used to associate new 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, files that use a different file system than the default file system can be used.

In order to accomplish 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.

➤ Example

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

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


5.4 Using the Environment Variable

In order to allow the file system to be setup during execution of a program, specify the following in the COBOL code. The (*) is only used for the ACUCOBOL runtime. Also, be aware that specification of a file system is usually done in the runtime configuration file and NOT changed in the COBOL program.

NOTE *Refer to the ACUCOBOL-GT, User's Manual (chapter 2.1 and 2.2) for detailed instructions on how to use the ACUCOBOL-GT compiler and runtime.*

➤ Syntax 1

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

➤ Syntax 2

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


6 Configuration File Variables

This section lists the acceptable ranges of data for DCI, as well as tables specifying how COBOL data types are mapped to DBMaker data types. Configuration file variables are used to modify the standard behavior of DCI and are stored in a file called DCI_CONFIG.

6.1 Setting DCI_CONFIG Variables

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 display an error and assumes that no configuration variables have been assigned. This variable is set in the COBOL runtime configuration file.

➤ Syntax 1

In Unix, DCI will look for the file DCI_CONFIG. This environment variable is used to establish the path and name of the DCI configuration file. Working with the Bourne shell, the following command can be used.

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

➤ Syntax 2

In DOS, DCI reads the configuration file called DCI_CONFIG in the directory c:\etc\test.

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

➤ Syntax 3

In UNIX, DCI utilizes the file called “DCI” in the directory /home/test.

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

DCI_CASE

File names in COBOL are case insensitive, however table names are case sensitive. This configuration variable determines how file names are translated into table names. Setting this configuration variable to *lower* means that file names are translated into table names with all lowercase characters. Setting this configuration variable to *upper* means that file names are translated into table names with all uppercase characters. Setting this configuration variable to *ignore* means that file names will not be translated into table names with all lowercase or

uppercase characters. The default setting for DCI_CASE is *lower*. If your file names are DBCS words, set DCI_CASE to *ignore*.

➤ **Example:**

```
DCI_CASE IGNORE
```

DCI_COMMIT_COUNT

The DCI_COMMIT_COUNT configuration variable indicates the conditions under which a COMMIT WORK operation is issued. There are two possible values, 0 and <*n*>.

DCI_COMMIT_COUNT=0

No automatic commit is done (default value).

DCI_COMMIT_COUNT=<*N*>

Under this condition DCI waits until the number of WRITE, REWRITE, AND DELETE operations are equal to the value <*n*> before issuing a COMMIT WORK statement. This rule is applied just if the file is open in “output” or “exclusive” mode.

DCI_DATABASE

DCI_DATABASE is used to specify the name of the database established during the setup of DBMaker.

➤ **Example 1**

The following entry has to be included in the configuration file if the database used is named DBMaker_Test.

```
DCI_DATABASE DBMaker_Test
```

➤ **Example 2**

Sometimes, the database name is not known in advance, and for this reason it is necessary to set it dynamically during 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"
```

➡ **Example 3**

Sometimes we want to access a table on a different database. You can use DCI_DATABASE to connect to more than one database and dynamically switch between databases.

```
* connect to DBSAMPLE4 to access idx-1-file
CALL "DCI_SETENV" USING "DCI_DATABASE" "DBSAMPLE4"
....
open output idx-1-file
....
* connect to DCIDB to access idx-2-file
CALL "DCI_SETENV" USING "DCI_DATABASE"
"DCIDB"
....
open output idx-2-file

* to switch dynamically to DBSAMPLE4 connection
CALL "DCI_SETENV" USING "DCI_DATABASE" "DBSAMPLE4"
close idx-1-file
...
```

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. 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_DEFAULT_RULES

Default management methods for the WHEN directive located in multi-definition files. The BEFORE statement indicates that a table be open when the \$WHEN condition is matched. The POST statement indicates that all related tables be open when the COBOL application opens multi-definition files.

The possible values are:

POST or COBOL

BEFORE or DBMS

DCI_DEFAULT_TABLESPACE

This variable is used to set the default tablespace where new tables are to be stored. The tablespace specified must already exist in the database. If no tablespace is specified by this variable, then new tables will be created in the default user tablespace.

DCI_DUPLICATE_CONNECTION

DCI_DUPLICATE_CONNECTION is used to acquire a lock when opening the same table and locking the same record two times in the same COBOL application by opening the same table using the same COBOL process but with a different database connection.

The default value is off (0).

➤ Example

To allow a COBOL application to acquire a lock on a table using different database connections:

```
DCI_DUPLICATION_CONNECTION 1
```

DCI_GET_EDGE_DATES

DCI_SET_EDGE_DATE is used to specify the value to be displayed if a user enters a low/high value in the DATE field. When a user inputs low/high value for a DATE field in a COBOL program, for example, by entering 00010101/99991231, the date will be displayed using COBOL's low/high value

00000000/99999999. When this variable is used, the low/high value of the DATE field will be displayed using the database's low/high value 00010101/99991231. This rule is also applied when the DATE field is a part of a key. The default value is off.

➤ **Syntax:**

The following line must be added in the dci.cfg file:

```
DCI_GET_EDGE_DATES 1
```

DCI_INV_DATE

This variable is used to establish an invalid date (like 2000/02/31) in order to avoid problems that can occur when an incorrect date format has been written to the database. The default for this variable is 99991230 (December 30th, 9999).

DCI_LOGFILE

This variable specifies the pathname of the DCI log file used to write all of the I/O operations executed by the interface. The *dci_trace.log* log file, stored in the */tmp* directory is used for debugging purposes. The use of a log file slows down the performance of DCI. For this reason it is recommended not add this variable in the configuration file unless deemed absolutely necessary.

➤ **Example**

A sample log file entry in the Config.ini file:

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

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 the JDBC Tool, or dmSQL.

NOTE *For more detailed information on creating new users, refer to the *JDBA Tool User's Guide* or the *Database Administrator's Guide*.*

➞ Example

A sample username entry, JOHNDOE, made in the Config.cfg 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.

One usage of this variable could be a COBOL program that uses dates from 1850 onwards. These dates can be stored in a database by setting the DATE directive to \$XFD DATE=JJJJJ (the date field must have the same number of characters) and setting the DCI configuration variable DCI_JULIAN_BASE_DATE to 18500101.

DCI_LOGTRACE

This variable sets different levels for the trace log.

- 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, one XFD can be used in conjunction with multiple files. A “*pattern*” can be made up of any valid filename characters. It may include the wildcard “*” symbol, which stands for any number of characters, or the question mark “?” , which stands for a single occurrence of any one character and can be used multiple times.

➤ **Syntax**

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

➤ **Example 1**

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

➤ **Example 2**

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_MAPPING CUST????=CUST
```

DCI_MAX_ATTRS_PER_TABLE

A DBMaker table may only have up to 252 columns. A COBOL file with more than 252 fields will not be able to map all fields to columns in the table. DCI provides the DCI_MAX_ATTRS_PER_TABLE configuration variable to define the number of fields at which the table will be split into two or more distinct tables. The multiple resulting tables must have unique names, so DCI appends the table name with an underscore (_) character followed by letters in consecutive order (A, B, C, etc.).

➤ **Example 1**

A COBOL file has 300 fields, and the following statement:

```
SELECT FILENAME ASSIGN TO "customer"
```

➤ **Syntax**

The following line must be added in the **dci.cfg** file:

```
DCI_MAX_ATTRS_PER_TABLE = 100.
```

➤ **Example 2**

Three tables will be created with the following names:

```
customer_a  
customer_b  
customer_c
```

DCI_MAX_BUFFER_LENGTH

DCI_MAX_BUFFER_LENGTH is used to split a cobol data record into multiple database tables, similar to the function performed by DCI_MAX_ATTRS_PER_TABLE. However, the cutoff value used to determine where a table will be split is determined by buffer length. The default value is 4096.

➤ Example 1

A COBOL record size contains 9000 bytes of data, and the following statement:

```
SELECT FILENAME ASSIGN TO "customer"
```

➤ Syntax:

The following line must be added in the dci.cfg file:

```
DCI_MAX_BUFFER_LENGTH 3000
```

➤ Example 2

Three tables will be created with the following names:

```
customer_a  
customer_b  
customer_c
```

DCI_MAX_DATE

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_DATE

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_NULL_ON_ILLEGAL_DATE

DCI_NULL_ON_ILLEGAL_DATE determines how COBOL data that is considered illegal by the database will be converted before it is stored. The value

1 causes all illegal data (except key fields) to be converted to null before it is stored. The value 0 (default value) causes the following conversions to occur:

- Illegal LOW-VALUES: stored as the lowest possible value (0 or - 99999...) or DCI_MIN_DATE default value.
- Illegal HIGH-VALUES: stored as the highest possible value (99999...) or DCI_MAX_DATE default value.
- Illegal SPACES: stored as zero (or DCI_MIN_DATE, in the case of a date field).
- Illegal DATE values: stored as DCI_INV_DATE default value.
- Illegal TIME: stored as DCI_INV_DATE default value.
- Illegal data in key fields is always converted, regardless of the value of this configuration variable.

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.

➡ Example 1

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
```

➡ Example 2

A password can also be accepted from a 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,

➡ Syntax 1

This statement can be used in the COBOL program to write or update the environment variable.

```
CALL "DCI_SETENV" USING "environment variable", value.
```

➤ Syntax 2

This statement can be used in the COBOL program to read the environment variable.

```
CALL "DCI_GETENV" USING "environment variable", value.
```

DCI_STORAGE_CONVENTION

This variable sets the COBOL storage convention. There are four value types currently supported by DBMaker.

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.

➤ **Example 1**

The following line is equal to; /usr/test/01/clients 01clients

DCI_USEDIR_LEVEL 1

➤ **Example 2**

The following line is equal to; /usr/test/01/clients test01clients

DCI_USEDIR_LEVEL 2

➤ **Example 3**

The following line is equal to; /usr/test/01/clients usrtest01clients

DCI_USEDIR_LEVEL 3

DCI_USER_PATH

When DCI looks for a file or files, the variable DCI_USER_PATH allows for specification of a username, or names. The user argument can be a period (.) with regard to the files, or the name of a user on the system.

➤ **Syntax**

DCI_USER_PATH user1 [user2] [user3] .

The type of OPEN statement issued for a file will determine the results of this setting.

OPEN STATEMENT	DCI_USER_PATH	DCI SEARCH SEQUENCE	RESULT
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.
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.
OPEN OUTPUT	Yes or no	Doesn't search for a user.	A new table will be made for the name associated with DCI_LOGIN.

Figure 6-1 Types of OPEN Statements

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.

➤ Example 1

Include the following entry in the configuration file in order to store data dictionaries in the directory */usr/DBMaker/Dictionaries*.

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

➤ Example 2

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

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

➤ Example 3

In a WIN-32 environment, “embedded spaces” can be specified using double-quotes.

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

<filename>_RULES

Default management for a multi-definition file. The actual file name replaces <filename>.

➤ Example

All of the files will use the POST rule except for the CLIENT file when the following commands are used.

```
DCI_DEFAULT_RULES      POST
CLIENT_RULES           BEFORE
```

DCI TABLE CACHE Variables

By default, DCI pre-reads data into the client data buffer to reduce client/server network traffic. The default maximum pre-read buffer is the smaller of 8kb/(record size) or 5 records.

It is possible that user's application will read a small table and only read a few records which are less than 8kb/(record size). For example, for a table with an

average record size of 20 bytes and a total of 1000 records, DBMaker will be able to read about 400 records (8kb/20) but the user's application may only read 4 or 5 records then call the START statement again. In this case, set the following variable to reduce the cache size and improve performance. Consider the application and data's behavior carefully when using these variables, or it may increase network traffic and cause reductions in performance.

The following are the three DCI_CACHE variables to set in the DCI_CONFIG file:

- **DCI_DEFAULT_CACHE_START** – sets the first read records to cache for START or READ. The default is the maximum of 8kb/(record size) or 5 records.
- **DCI_DEFAULT_CACHE_NEXT** – sets the next read records after the first cached record for START or READ have been read or discarded. The default is the maximum of 8kb/(record size) or 5 records.
- **DCI_DEFAULT_CACHE_PREV** – sets the read records for caching the previous records after the first cache record for START or READ have been read or discarded.

The default is $DCI_DEFAULT_CACHE_NEXT/2$.

Setting these variables in the DCI_CONFIG will affect all the tables in the user's application.

➤ **Example:**

DCI_DEFAULT_CACHE_START	10
DCI_DEFAULT_CACHE_NEXT	10
DCI_DEFAULT_CACHE_PREV	5

DCI_TABLESPACE

This allows you to define in which tablespace to create a table. It also works with wildcards. It is important only when a table is first created. Once the table exists, DCI does not monitor the value of this variable.

➤ **Example 1:**

You want to create the customer table in tablespace tbs1:

```
DCI_TABLESPACE customer=tbs1
```


➤ Example 2:

You want to create all tables that begin with cust in tablespace tbs1.

```
DCI_TABLESPACE cust*=tbs1
```

DCI_AUTOMATIC_SCHEMA_ADJUST

This variable directs DCI to alter the table schema definition when the XFD differs from the table schema. This variable is incompatible with the split tables (those with a number of columns > 250, and those whose record size is greater than 4 KB -exclude the BLOB field).

The possible values of this variable are:

- 0 Default, does nothing
- 1 Add the new fields to the table, and drop the ones who are not in the XFD
- 2 Add the new fields to the table, but do not drop the ones who are not in the XFD

DCI_INCLUDE

This variable permits the inclusion of an additional DCI_CONFIG file. It works as the COBOL COPY statement, and allows you to define more complex configurations.

➤ Example:

```
DCI_INCLUDE /etc/generic_dci_config
```

DCI_IGNORE_MAX_BUFFER_LENGTH

This variable is used to ignore the setting of DCI_MAX_BUFFER_LENGTH value. It will not split the table when the record length > 4k. The default is off.

DCI_NULL_DATE

When DCI writes a date field with this value it will write NULL, and when DCI reads a date with a NULL value, it will return DCI_NULL_DATE to a COBOL program.

DCI_NULL_ON_MIN_DATE

With this variable set to 1 the following action occurs. When a COBOL program writes a value of 0 to a DATE field, the value is stored in the database as NULL. Likewise, when a NULL value is read from the database the COBOL FD will be 0.

DCI_DB_MAP

This variable is used to map files in different directories as tables of different databases. Refer to Mapping to Multiple Databases for more info.

DCI_VARCHAR

With this variable set to 1 the following action occurs: When a COBOL program creates a new table (trought OPEN OUTPUT verb) all fields that were created as CHAR will become VARCHAR.

DCI_GRANT_ON_OUTPUT

This new option allows you to specify a GRANT command after table creation (OPEN OUTPUT)

➡ Example:

To grant select and delete privileges to user1 after any table creation:

```
DCI_GRANT_ON_OUTPUT SELECT,DELETE user1
```

7 DCI Functions

This section lists the DCI functions that could be called in the COBOL program. To enable these functions, user need to add these functions in the sub85.c and rebuild the DCI runtime.

7.1 Calling DCI functions

You can call these DCI functions by writing:

```
CALL "dci_function_name" USING variable [, variable, ...]
```

in your COBOL program.

DCI_SETENV

This function is used to write or update the environment variable.

➤ **Syntax:**

```
CALL "DCI_SETENV" USING "environment variable", value
```

➤ **Example**

```
call "DCI_SETENV" using "DCI_DATABASE" , "DBSAMPLE4"
```

DCI_GETENV

This function is used to read the environment variable.

➤ **Syntax**

```
CALL "DCI_GETENV" USING "environment variable", variable
```

➤ **Example**

```
CALL "DCI_GETENV" USING "DCI_DATABASE", ws_dci_database
```

DCI_DISCONNECT

This function is used to disconnect a database connection.

➤ **Example 1**

If there is only one connection in the cobol program, use the following code to disconnect from the database.

```
CALL "DCI_DISCONNECT".
```

➤ **Example 2**

If there is more than one connection the COBOL program, use the following code to disconnect a specific database.

```
CALL "DCI_DISCONNECT" USING "DBSAMPLE4"
```

DCI_GET_TABLE_NAME

This function is used to get the table name of the passed COBOL name (It's not always so immediate to know the effective table name, because there can be some manipulation in these cases: XFD WHEN ... TABLENAME.).

```
CALL "DCI_GET_TABLE_NAME" USING ws-filename, ws-dci-file-name
```

DCI_SET_TABLE_CACHE

This function is used to dynamically change the cache for tables set these variables before START or READ statements.

☞ Example:

```
...
WORKING-STORAGE SECTION.
    01 CACHE-START PIC 9(5) VALUE 10.
    01 CACHE-NEXT  PIC 9(5) VALUE 20.
    01 CACHE-PREV  PIC 9(5) VALUE 30.
...
PROCEDURE DIVISION.
    OPEN INPUT IDX-1-FILE
    MOVE SPACES TO IDX-1-KEY
    CALL "DCI_SET_TABLE_CACHE" USING CACHE-START
                                CACHE-NEXT
                                CACHE-PREV
    START IDX-1-FILE KEY IS NOT LESS IDX-1-KEY.
    PERFORM VARYING IND FROM 1 BY 1 UNTIL IND = 10000
        READ IDX-1-FILE NEXT AT END EXIT PERFORM END-READ
        DISPLAY IND AT 0101
    END-PERFORM
    CLOSE IDX-1-FILE
```

DCI_BLOB_ERROR

This function is used to get the error after calling DCI_BLOB_GET or DCI_BLOB_PUT.

➤ Example:

```

working-storage section.
77  BLOB-ERROR-ERRNO pic S9(4) COMP-5.
77  BLOB-ERROR-INT-ERRNO pic S9(4) COMP-5.

PROCEDURE DIVISION.
    CALL "DCI_BLOB_ERROR" USING BLOB-ERROR-ERRNO
                                BLOB-ERROR-INT-ERRNO
    DISPLAY "BLOB-ERROR-ERRNO=" BLOB-ERROR-ERRNO.
    DISPLAY "BLOB-ERROR-INT-ERRNO=" BLOB-ERROR-INT-ERRNO.
    
```

DCI_BLOB_GET

This function is used to give users more effectively use of BLOB data in a COBOL program. By using the DCI_BLOB_GET command you can quickly and efficiently access BLOB data using COBOL. When using the DCI_BLOB_GET command you must follow the rules listed below:

- The user's table must have a BLOB (long varchar/long varbinary) data type
- Users cannot set the field with BLOB type in the COBOL FD
- Users can only use the DCI_BLOB_GET command after the READ, READ NEXT or READ PREVIOUS command

➤ Example

A user creates a table by:

```

CREATE TABLE BLOBTB (
    SB_CODCLI  char(8),
    SB_PROG    SERIAL,
    IL_BLOB    LONG VARBINARY,
    PRIMARY KEY ("sb_codcli")) LOCK MODE ROW NOCACHE;
    
```

The following gives a practical application of the use of the DCI_BLOB_GET in the COBOL program..

```

identification division.
program-id. blobtb.
    
```

```

date-written.
remarks.
environment division.
input-output section.
file-control.
    SELECT BLOBTB ASSIGN TO RANDOM, "BLOBTB"
        ORGANIZATION IS INDEXED
        ACCESS IS DYNAMIC
        FILE STATUS IS I-O-STATUS
        RECORD KEY IS SB-CODCLI.

*=====*
data division.
file section.
FD BLOBTB.
01 SB-RECORD.
    03 SB-CODCLI          PIC X(8).
    03 SB-PROG            PIC S9(9) COMP-5.

*=====*
working-storage section.
77 I-O-STATUS pic xx.
77 BLOB-ERROR-ERRNO pic S9(4) COMP-5.
77 BLOB-ERROR-INT-ERRNO pic S9(4) COMP-5.

procedure division.
main.
    open i-o blobtb
    initialize sb-record.
    READ blobtb next.
    CALL "DCI_BLOB_GET" USING "il_blob" "laecopy.bmp" 1.
    CALL "DCI_BLOB_ERROR" USIG BLOB-ERROR-ERRNO
        BLOB-ERROR-INT-ERRNO
    DISPLAY "BLOB-ERROR-ERRNO=" BLOB-ERROR-ERRNO.
    DISPLAY "BLOB-ERROR-INT-ERRNO=" BLOB-ERROR-INT-ERRNO.
    DISPLAY "SB-CODCLI=" SB-CODCLI.
    DISPLAY "SB-PROG=" SB-PROG.
    close blobtb.
    ACCEPT OMITTED.
    stop run.

```

DCI_BLOB_PUT

This function is used to enable users to more effectively use BLOB data in a COBOL program. Using the DCI_BLOB_PUT command you can insert data into a BLOB. When using the DCI_BLOB_PUT command you must follow the rules listed below:

- The user's table must have a BLOB (long varchar/long varbinary) data type.
- Users cannot set the field with BLOB type in the COBOL FD.
- Users can only call the DCI_BLOB_PUT command before a WRITE or REWRITE command.
 - If user does not call DCI_BLOB_PUT before a WRITE statement, the default value will be inserted in the blob column.
 - If user does not call DCI_BLOB_PUT before REWRITE statement, the blob column will not be updated.

☞ Example:

The following gives a practical application of the use of the DCI_BLOB_PUT in the COBOL program.

First the user creates a table:

```
CREATE TABLE BLOBTB (
    SB_CODCLI  char(8),
    SB_PROG    SERIAL,
    IL_BLOB    LONG VARBINARY,
    PRIMARY KEY ("sb_codcli")) LOCK MODE ROW NOCACHE;
```

Once the table is created the user continues with the following.

```
identification division.
    program-id. blobtb.
    date-written.
    remarks.
    environment division.
    input-output section.
    file-control.
        SELECT BLOBTB ASSIGN TO RANDOM, "BLOBTB"
            ORGANIZATION IS INDEXED
            ACCESS IS DYNAMIC
            FILE STATUS IS I-O-STATUS
            RECORD KEY IS SB-CODCLI.
```



```

=====*
data division.
file section.
FD BLOBTB.
01 SB-RECORD.
    03 SB-CODCLI          PIC X(8).
    03 SB-PROG            PIC S9(9) COMP-5.
=====*
working-storage section.
77 I-O-STATUS pic xx.
77 BLOB-ERROR-ERRNO pic S9(4) COMP-5.
77 BLOB-ERROR-INT-ERRNO pic S9(4) COMP-5.

procedure division.
main.
    open i-o blobtb
    move "AAAAAAA" TO SB-CODCLI.
    move 0 TO SB-PROG.
    CALL "DCI_BLOB_PUT" USING "il_blob" "laetitia.bmp".
    WRITE SB-RECORD.
    close blobtb.
    ACCEPT OMITTED.
    stop run.

```

DCI_GET_TABLE_SERIAL_VALUE

This function is used to get the serial value after a WRITE statement.

➞ Example:

```

FD SERIALTB.
01 SB-RECORD.
    03 SB-CODCLI          PIC X(8).
$XFD COMMENT dci serial
    03 SB-PROG            PIC S9(9) COMP-5.
working-storage section.
77 I-O-STATUS pic xx.
77 SERIAL-NUM pic S9(9) COMP-5.

```

```
procedure division.  
main.  
    open i-o serialtb  
    move "AAAAAAA" TO SB-CODCLI.  
    move 0 TO SB-PROG.  
    WRITE SB-RECORD.  
    CALL "DCI_GET_TABLE_SERIAL_VALUE" USING SERIAL-NUM.  
    DISPLAY "SERIAL-NUM=" SERIAL-NUM.
```

DCI_FREE_XFD

This function is used to purge the XFD image DCI keeps in cache. This can be useful, to reload a XFD that changed after a table has already been opened by this connection.

```
CALL "DCI_FREE_XFD"
```

8 COBOL Conversions

Transactions are enforced in DCI during conversions. 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.

DCI doesn't support record encryption, record compression, or the alternate collating sequence. If these options are included in code, they will be disregarded. DCI also doesn't support the "P" PICTURE edit function in the XFD data definition and all file names are converted to lowercase.

DBMAKER DATABASE SETTINGS	RANGE LIMIT
Indexed key size.	1024
Number of columns per key.	16
Length for a CHAR field.	3992 bytes
Simultaneous RDBMS connections.	1200
Character for column names.	32
Database tables simultaneously open by a single process.	256

Figure 8-1 DBMaker Database Settings Range Limits table

8.1 Using Special Directives

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

The DBMaker OID, VARCHAR(size), and FILE data types are not currently supported with special directives.

DBMAKER DATA TYPE	DIRECTIVE
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
SERIAL	Using XFD COMMENT DCI SERIAL

Figure 8-2 DBMaker Data Types Supported using Special Directives

8.2 Mapping COBOL Data Types

DCI establishes what it considers to be the best match for COBOL data types in the creation of all columns in a DBMaker database table. Any data the COBOL data type can contain can also be contained in the database column. The XFD directives that have been specified will be checked first.

COBOL	DBMAKER		COBOL	DBMAKER
9(1-4)	SMALLINT		9(5-9) comp-4	INTEGER
9(5-9)	INTEGER		9(10-18) comp-4	DECIMAL(10-18)
9(10-18)	DECIMAL(10-18)		9(1-4) comp-5	SMALLINT
s9(1-4)	SMALLINT		9(5-10) comp-5	DECIMAL(10)
s9(5-9)	INTEGER		s9(1-4) comp-5	SMALLINT
s9(10-18)	DECIMAL(10-18)		s9(5-10) comp-5	DECIMAL(10)
9(n) comp-1 n (1-17)	INTEGER		9(1-4) comp-6	SMALLINT
s9(n) comp-1 n (1-17)	INTEGER		9(5-9) comp-6	INTEGER
9(1-4) comp-2	SMALLINT		9(10-18) comp-6	DECIMAL(10-18)
9(5-9) comp-2	INTEGER		s9(1-4) comp-6	SMALLINT
9(10-18) comp-2	DECIMAL(10-18)		s9(5-9) comp-6	INTEGER
s9(1-4) comp-2	SMALLINT		s9(10-18) comp-6	DECIMAL(10-18)
s9(5-9) comp-2	INTEGER		signed-short	SMALLINT
s9(10-18) comp-2	DECIMAL(10-18)		unsigned-short	SMALLINT
9(1-4) comp-3	SMALLINT		signed-int	CHAR(10)
9(5-9) comp-3	INTEGER		unsigned-int	CHAR(10)
9(10-18) comp-3	DECIMAL(10-18)		signed-long	CHAR(18)
s9(1-4) comp-3	SMALLINT		unsigned-long	CHAR(18)
s9(5-9) comp-3	INTEGER		float	FLOAT
s9(10-18) comp-3	DECIMAL(10-18)		Double	DOUBLE
9(1-4) comp-4	SMALLINT		PIC x(n)	CHAR(n) n 1-max column length

Figure 8-3 COBOL to DBMaker Data Type Conversion Chart

8.3 Mapping DBMaker Data Types

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

It is not necessary to worry about exactly matching the database data types to COBOL data types. 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		DBMAKER	COBOL
SMALLINT	9(1-4)		INTEGER	9(5-9) comp-4
INTEGER	9(5-9)		DECIMAL(10-18)	9(10-18) comp-4
DECIMAL(10-18)	9(10-18)		SMALLINT	9(1-4) comp-5
SMALLINT	s9(1-4)		DECIMAL(10)	9(5-10) comp-5
INTEGER	s9(5-9)		SMALLINT	s9(1-4) comp-5
DECIMAL(10-18)	s9(10-18)		DECIMAL(10)	s9(5-10) comp-5
INTEGER	9(n) comp-1 n (1-17)		SMALLINT	9(1-4) comp-6
INTEGER	s9(n) comp-1 n (1-17)		INTEGER	9(5-9) comp-6
SMALLINT	9(1-4) comp-2		DECIMAL(10-18)	9(10-18) comp-6
INTEGER	9(5-9) comp-2		SMALLINT	s9(1-4) comp-6
DECIMAL(10-18)	9(10-18) comp-2		INTEGER	s9(5-9) comp-6
SMALLINT	s9(1-4) comp-2		DECIMAL(10-18)	s9(10-18) comp-6
INTEGER	s9(5-9) comp-2		SMALLINT	signed-short
DECIMAL(10-18)	s9(10-18) comp-2		SMALLINT	unsigned-short
SMALLINT	9(1-4) comp-3		CHAR(10)	signed-int
INTEGER	9(5-9) comp-3		CHAR(10)	unsigned-int
DECIMAL(10-18)	9(10-18) comp-3		CHAR(18)	signed-long
SMALLINT	s9(1-4) comp-3		CHAR(18)	unsigned-long
INTEGER	s9(5-9) comp-3		FLOAT	float
DECIMAL(10-18)	s9(10-18) comp-3		DOUBLE	Double
SMALLINT	9(1-4) comp-4		CHAR(n) n 1-max column length	PIC x(n)

Figure 8-4 DBMaker to COBOL Data Type Conversion Chart

8.4 Troubleshooting 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.

ERROR	DEFINITION	INTERPRETATION	SOLUTION
9D,01	There is a read error on the dictionary file.	An error occurred while reading the XFD file. The XFD file is corrupt.	Recompile with -Fx to re-create the dictionary file.
9D,02	There is a corrupt dictionary file. The dictionary file 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 a correct directory in the DCI_XFDPATH configuration file variable (it may be necessary to recompile using -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 of the "xxx" variable is illegal.	An elementary data item in an 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.	

Figure 8-5 DCI Secondary Errors Chart

8.5 Troubleshooting Native SQL Errors

Some native SQL errors may be generated by a database while using DCI for DBMaker. The exact error number and wording may vary from database to database.

NUMBER	DEFINITION	INTERPRETATION	SOLUTION
9D, 6523,6018	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 the NAME directive to the FD field of an invalid column and recompile to create a new XFD file.
9D, 1310	Journal full, command rolled back to internal savepoint		Add "start transaction/commit/rollback" code in the COBOL program. Or set DCI_COMMIT_COUNT in the DCI configuration file.
9D, 5503	invalid key name	The table does not have the index	Create the index with correct index name and columns
9D, 5504	Cannot use host variable		User cannot use host variable in the runsql.acu
9D, 5508	do not have INSERT/UPDATE/DELETE privilege	User cannot open I-O for table that they does not have the insert/delete/update privilege	OPEN INPUT with that table
9D, 5512	cannot issue select query		User cannot issue select statement in the runsql.acu
9D,5513	client-server version mismatch when dci connect	User's DCI runtime is newer than the dmserver	User should upgrade their dmserver before running new DCI runtime
9D, 5514	invalid column number	COBOL FD column number > table column number	User need to check the FD column number and table column number
9D, 5515	invalid XFD column name or data type and length does not match	COBOL FD column name or column type does not match with table definition	Compare the FD and table definition. Fix this problem by either change the COBOL FD or alter table.
9D, 5518	DCI blob data is null	When user get blob from a column and the data is null	

Figure 8-6 Native SQL Errors Char

8.6 Converting Vision Files

DCI provides a sample program to convert COBOL files into RDBMS tables. Before using the DCI_MIGRATE program, a Vision file to be converted and an XFD data dictionary for the Vision file are required. The ACUCOBOL runtime system 4.3 or higher linked to DCI must be installed and a DCI_MIGRATE object program must be ready.

Using DCI_Migrate

This is a general-purpose program that converts any COBOL vision file into a DBMaker table. To run correctly the minimum DCI configuration settings must be defined 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 DCI_MIGRATE 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 by reading DBMaker rows.

The DCI_MIGRATE program will report the following:

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

DCI_MIGRATE OPTIONS	RESULT
--help	Displays the online help.
--nowait	Doesn't wait for user confirmation during interactive mode.
--noverify	Skips the verify process.
--nomigrate	Skips the migrate process.
--visdbm	Converts vision files to DBMaker tables (default).
--dbmvis	Converts DBMaker tables to vision files.

Figure 8-7 DCI_MMIGRATE Options Result table

☛ **Syntax 1**

The *vision_file_name* is the name of the Vision file to be converted and *dbm_table_name* is the name of the DBMaker table.

```
runcbl DCI_MIGRATE vision_file_name dbm_table_name [options]
```

☛ **Syntax 2**

Setting the environment variable named DCI_MIGRATE to “yes” can turn off the report. The report will then append a file named “dbm_table_name.log”.

```
DCI_MIGRATE = yes
```

☛ **Syntax 3**

The record can be dumped for 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 an interface between an application and an operating system.

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 in the same way as ordinary records. The database can only access the name and location of a BLOB; typically, another application is used to read the data.

Buffer

A buffer is an internal memory space (zone) where data is temporarily stored during input or output operations.

Client

A computer that can access and manipulate data that is stored on a central server computer.

Column

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

Data dictionaries

Also known as extended file descriptors; they serve as maps (links) between database schema and the file descriptors in a COBOL application.

Directive

An optional comment placed in the COBOL code that sets the proceeding 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

Files containing a list of keys that uniquely identify all records.

Key

A unique value used to identify a record in a database. (See ***Primary Key*** for more details.)

Primary key

A primary key consists of a column of unique (or key) values , which can be used to identify individual records contained in a table.

Query

In DBMaker, SQL commands used to execute data query requests made by a user to obtain specific information.

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 table columns.

Relational Database

A relational database is a database system where internal database tables on different databases may be related to one another by the use of 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 server is a central computer that stores and handles network configuration files, which also can consist of a database management system to store data (database) and distribute data to clients via a network connection.

SQL

Structured Query Language: The language which DBMaker and other ODBC compliant programs use to access and manipulate data.

Table

A logical storage unit in a database that consists of columns and rows used to store records.

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, 4-3

B

BINARY Directive, 4-4

B-TREE

Files, 1-1

C

Column Names

Maximum Length, 8-2

Columns, 3-5

COMMENT Directive, 4-4, 4-5

Configuration

Basic, 2-14

Configuration file variables, 6-1

Configuration File Variables

_DCI_MAPPING, 6-7

CDI_LOGFILE, 6-6

DCI Table Cache, 6-13

DCI_AUTOMATIC_SCHEMA_ADJUST, 6-15

T, 6-15

DCI_DATABASE, 6-3

DCI_DATE_CUTOFF, 6-4

DCI_DB_MAP, 6-16

DCI_IGONRE_MAX_BUFFER_LENGTH, 6-15

H, 6-15

DCI_INCLUDE, 6-15

DCI_INV_DATE, 6-6

DCI_JULIAN_BASE_DATE, 6-7

DCI_LOGIN, 6-6

DCI_LOGTRACE, 6-7

DCI_MAX_DATE, 6-9

DCI_MIN_DATE, 6-9

DCI_NULL_DATE, 6-15

DCI_NULL_ON_MIN_DATE, 6-16

DCI_PASSWD, 6-10

DCI_STORAGE_CONVENTION, 6-11

DCI_TABLESPACE, 6-14

DCI_USEDIR_LEVEL, 6-11

DCI_USER_PATH, 6-12

DCI_XFDPATH, 6-13

DEFAULT_RULES, 6-5

filename_RULES, 6-13

D

Data Dictionaries

Storage Location, 6-13

Data Structures, 2-2

Data Types

COBOL to DBMaker, 8-3

DBMaker to COBOL, 8-5

Not Supported, 8-2

Supported, 8-2

Database Name

Specifying, 6-3

DATE Directive, 4-6

DCI_CONFIG, 6-1

DCI_DATABASE, 6-3

DCI_DATE_CUTOFF, 6-4

DCI_INV_DATE, 6-6

DCI_JULIAN_BASE_DATE, 6-7

DCI_LOGFILE, 6-6

DCI_LOGIN, 6-6

DCI_LOGTRACE, 6-7

DCI_MAPPING, 3-14, 6-7

DCI_MAX_DATE, 6-9

DCI_MIN_DATE, 6-9

DCI_PASSWD, 6-10

DCI_STORAGE_CONVENTION, 6-11

DCI_USEDIR_LEVEL, 6-11

DCI_USER_PATH, 6-12

DCI_XFDPATH, 6-13

Default Filing System, 5-3

DEFAULT_RULES, 6-5

DEFAULT-HOST setting, 5-2

Directives, 4-1

ALPHA, 4-3

BINARY, 4-4

COMMENT, 4-4, 4-5

DATE, 4-6

FILE, 4-9

NAME, 4-9

NUMERIC, 4-10

Supported, 4-3

Syntax, 4-2

USE GROUP, 4-10

VAR-LENGTH, 4-11

WHEN, 4-12

Document Conventions, 1-5

E

embedded SQL, 1-1

Errors

Runtime, 8-7

SQL, 8-9

Extended File Descriptors, 3-1

F

Field Names

Identical, 3-7

Long, 3-8

FILE CONTROL section, 3-2

FILE Directive, 4-9

File System, 2-2

FILE=*Filename* Directive, 4-9

filename_RULES(*), 6-13

Filing System Options, 5-2

FILLER data items, 3-13

I

I/O Statements, 1-1
Illegal DATE values, 2-18, 6-10
Illegal HIGH-VALUES, 2-18, 6-10
Illegal LOW-VALUES, 2-18, 6-10
Illegal SPACES, 2-18, 6-10
Illegal time, 2-18, 6-10
Invalid Data, 2-18

J

Julian dates, 4-7

K

Key fields, 3-5
KEY IS phrase, 3-5, 3-12

L

Login, 6-6

M

Multiple Record Formats, 3-9

N

NAME Directive, 4-9
NUMERIC Directive, 4-10

O

OCCURS Clauses, 3-13

P

Password, 6-10

Platforms

Supported, 2-5
Primary Keys, 3-5

R

Records, 3-5
REDEFINES Clause, 3-12
Requirements
 Software, 2-5
 System, 2-5
Runtime Configuration File, 2-6, 2-7, 2-10
Runtime Errors, 8-7
Runtime Options, 5-1

S

Sample Application, 2-20
Schema, 3-10
SELECT statement, 3-2, 3-6
Setup, 2-6
 UNIX, 2-9
 Windows, 2-6
Shared Libraries, 2-13
Software Requirements, 2-5
Sources of Information, 1-3
SQL
 Embedded, 1-1
 Errors, 8-9
Supported Features, 8-1
Supported Platforms, 2-5
System Requirements, 2-5

T

Table Schema, 3-10

Tables, 3-2

Technical Support:, 1-4

U

USE GROUP Directive, 4-10

User Name, 6-6

V

VAR-LENGH Directive, 4-11

Vision file system, 5-2

W

WHEN Directive, 4-12

X

XFD files, 3-1