CRISP® Datagate User's Guide

SQUARE D COMPANY
CRISP AUTOMATION SYSTEMS

CRISP®
Datagate User's Guide

Dedicated to Growth
Committed to Quality

CRISP®
Datagate User's Guide

Dedicated to Growth
Committed to Quality

Datagate *User's Guide*

SQUARE D COMPANY
CRISP AUTOMATION SYSTEMS

Datagate User's Guide

Document number: 500 046 - 001, Rev.

Document History

Revision	Date	Pages affected/Description of change
a	8/9/90	
b	8/16/90	
c	9/25/90	
d	12/6/90	

Software Version CRISP/32 Rev. 2.7 and later

Table of Contents

Configuring Datagate (cont)

Opcode Definitions

Opcode Definitions (cont)

Introduction

Introduction

The DATAGATE User's Guide provides the information necessary for you to operate the DATAGATE software.

This manual contains the following sections.

Section	**Description**
Operation *(page 3)*	This section of the manual contains general information concerning the basic operation of the DATAGATE software (e.g., Product Design Philosophy, VAX implementation, etc.).
Installing DATAGATE *(page 5)*	This section of the manual contains information concerning the software installation, configuration, starting/stopping DATAGATE, etc.
Configuring DATAGATE *(page 11)*	This section of the manual contains detailed information concerning the configuration of the DATAGATE software.
Opcode Definitions *(page 49)*	This section of the manual contains detailed information concerning the valid DATAGATE opcodes.
Token Definitions *(page 69)*	This section of the manual contains detailed information concerning keywords that may be used in place of literal numeric values.
Status and Error Reporting *(page 73)*	This section of the manual contains detailed information concerning the codes that will be returned to the application database.
Glossary *(page 77)*	This section of the manual contains information concerning terms, abbreviations, etc. used in this manual.

Notes:

Operation

General

The DATAGATE User's Guide provides the information concerning the operation of the DATAGATE software.

This section contains the following subsections.

Section	Description
Product Design Philosophy *(page 4)*	This section of the manual contains general information concerning the basic operation of the DATAGATE software (e.g., Product Design Philosophy, VAX implementation, etc.).
CRISP VAX Implementation *(page 4)*	This section of the manual contains information concerning the implementation of DATAGATE in the CRISP/VAX environment.

Product Design Philosophy

DATAGATE is a software product that is designed to handle the transfer of data between any Relational Database Management System supporting a Dynamic SQL interface and any host computer database. RDBMS users are provided with both a programmatic means to transfer data (used in conjunction with the embedded SQL capabilities of the RDBMS product) and a means to setup configured data transfers. The intent of a configured transfer is to allow users to specify SQL statements which contain the names of application database variables, whose values are to take the place of values in the SQL statement. DATAGATE is designed to allow totally flexible bi-directional data transfers.

SQUARE D/CRISP Automation is continually improving this product. If you do not see a relational database construct that you need, please contact us. One of our DATAGATE experts would be happy to address your needs specifically.

CRISP VAX Implementation

The specific DATAGATE implementation described in this manual is used to transfer data between any number of Relational Database Management Systems (RDBMS)and CRISP VAX databases. For configured transfers, the user simply places a description of the desired configuration in a disk file using DATAGATE configuration commands in combination with ANSI Standard Query Language (SQL) statements with a special syntax to denote CRISP Application Database (ADB) variables. DATAGATE will then automatically perform the data transfers between the tables specified and the CRISP databases.

The parts of the user configuration file are described in the Configuring DATAGATE section. They include a description of the commands, command defaults, and the relationship between relational database tables and the CRISP databases.

For programmatic transfers, users will write their own Fortran or "C" program containing both embedded SQL statements (for Rdb access) and calls to a DATAGATE Run-Time Library (RTL) Communications package which allows easy transfer of data into and out of CRISP databases on any node.

Installing DATAGATE

General

The DATAGATE User's Guide provides the information concerning the installation of the DATAGATE software.

This section contains the following subsections.

Section	Description
Software Installation *(page 6)*	This section of the manual contains general information concerning the installation of the DATAGATE software.
Configuring DATAGATE *(page 8)*	This section of the manual defines the configuration choices available depending on whether or not DATAGATE is installed on a system with CRISP/32.

Software Installation

This software is installed on your VAX/VMS system using Digital Equipment Corporations standard VMSINSTAL.COM. Refer to the installation guide provided with DATAGATE for more detailed installation information. Please read this entire section before proceeding with the installation.

For performance reasons, it is preferrable if the RDBMS of the user's choice is installed on a non-CRISP VAX. Then, DATAGATE is installed on the same VAX as the RDBMS. However, DATAGATE may be installed on a CRISP VAX as long as the RDBMS is also installed on that VAX.

In order for DATAGATE to function, WORF and DATAGATE must be installed. The order in which the two products are installed is not important; however, both products must be on the DATAGATE VAX before attempting to start DATAGATE.

DGT Installation

Most of the DATAGATE files are installed in the directory [TAG.DGT] on the specified disk; however, the DATAGATE run-time libraries are installed in SYS$SHARE and a startup procedure is placed in SYS$STARTUP on the system disk.

The following files are installed.

```
user$disk:[TAG.DGT]  DGT.EXE
                     DGT_BUILD_INGRES_C_EXAMPLE.COM
                     DGT_BUILD_RDBVMS_C_EXAMPLE.COM
                     DGT_C_EXAMPLE.TEMPLATE
                     DGT_RDBVMS.UCF
                     DGT_SAMPLE.UCF
                     DGT_USERMAN.MEM
                     DGTTST.C32
                     USER_CONFIG_DGT.COM*
                     UCP$TRD.EXE
                     UCP$TIK.EXE
                     UCP$UCP.EXE

SYS$SHARE            CRISPDGTRDBVMSRTL.EXE
                     CRISPDGTINGRESRTL.EXE

SYS$STARTUP          DGT_STARTUP.COM

CRISP$               USER_CONFIG_DGT.COM*
```

In addition to the previous files, VMS help shall be available by typing the following.

```
   $ HELP DGT
```

(Continued on next page.)

+ **Note:**

The sample user configuration files (UCFs) provide with this product are intended as a reference only. Do not give your production UCF the same name as any of the example UCFs or they will be overwritten by subsequent DATAGATE upgrades. Also, the location of USER_CONFIG_DGT.COM is dependent on whether DATAGATE is installed on a CRISP or a non-CRISP VAX.

Once DATAGATE has been installed using VMSINSTAL, execute the following.

```
   $ @SYS$STARTUP:DGT_STARTUP ddc0: rdbms_1 [rdbms_2]
```

Where `ddc0:` should be replaced by the device that DATAGATE was installed on and where `rdbms_1` and, optionally, `rdbms_2` should be replaced by one of the tokens RDBMVS or INGRES depending on the relational database that you will use. You will only need to replace `rdbms_2` is you have both Rdb/VMS and Ingres on your system.

This procedure creates the TAG_DGT$DEVICE logical and install the DATAGATE run-time libraries. If this installation is performed on a cluster, you must execute the previous line on all other nodes in the cluster.

In addition, you will need to add the previous line to the system startup file (SYS$STARTUP:SYSTARTUP_V5.COM).

WORF Installation

The WORF files are all installed on the system disk. They are installed as follows.

```
SYS$LIBRARY:          CRISPRTL.EXE
                      CRISPWORFRTL.EXE
                      CRISPUSERLIB.TLB

SYS$STARTUP:          WORF_STARTUP.COM

SYS$EXAMPLES:         WORF_EXAMPLE_C.C
                      WORF_EXAMPLE_FOR.FOR

SYS$HELP:             WORF*.RELEASE_NOTES
```

(Continued on next page.)

WORF Installation (cont)

When the WORF installation is complete, you must execute the following line and add it to the system startup procedure (SYS$MANAGER:SYSTARTUP_V5.COM).

```
$ @SYS$STARTUP:WORF_STARTUP ddc0: [ddc0:]
```

Where `ddc0:` should be replaced by the device name(s) of the network devices (e.g., `XQA0:`) that WORF is to use for communicationns. The startup procedure defines system logical names for CRISP$NET00 and CRISP$NET01 to refer to the device(s)

If this installation is performed on a cluster, execute the previous line on all other nodes in the cluster.

Configuring DATAGATE

Once DATAGATE is installed, it must be configured. The configuration choices available are dependant on whether DATAGATE is installed on a CRISP or a non-CRISP VAX.

Configuring on a CRISP VAX

If DATAGATE is installed on a CRISP VAX, execute the following command procedure.

```
$ @[CRISP]USER_CONFIG_DGT
```

This command procedure initially prompts you for the name of the disk where the product has been installed, if it was unable to determine this information automatically. This would normally be the disk that holds you RDBMS or possibly your system disk. You are also prompted to supply the name of your user configuration file (UCF), which is the file that describes your specific requirements. You must enter the full file specification, including node, device, and directory.

You are then prompted whether you want DATAGATE to run DETACHED or on a BATCH queue. This is dependant on you specific system requirements.

If you specify that DATAGATE should run in BATCH mode, you are then prompted for the name of the batch queue and you are prompted to verify that the job limit qualifier for the queue is large enough to run DATAGATE in addition to other system batch jobs.

(Continued on next page.)

Configuring on a CRISP VAX (cont)

Regardless of whether you choose to run DATAGATE as DETACHED process or on a batch queue, you are prompted for the priority for each DATAGATE process. The recommended value is 4.

The results of the configuration process are written to [CRISP]USER_START_DGT.COM. DATAGATE will start automatically when CRISP is started.

Configuring on a non-CRISP VAX

If DATAGATE is installed on a non-CRISP VAX, execute the following command procedure.

```
$ @[TAG.DGT]USER_CONFIG_DGT
```

This command procedure initially prompts you for the name of the disk where the product has been installed, if it was unable to determine this information automatically. You are also prompted to supply the name of your user configuration file (UCF), which is the file that describes your specific requirements. You must enter the full file specification, including node, device, and directory.

You are then prompted for the name of the batch queue that DATAGATE is to run on. Enter the name of a batch queue on your system. You are prompted to verify that the job limit qualifier for the queue is large enough to run DATAGATE in addition to your other system requirements.

Finally, you are prompted for the priority at which DATAGATE should run. The recommended value is 4.

The results of the configuration process are written to [TAG.DDGT]USER_START_DGT.COM.

Starting DATAGATE

To start DATAGATE on a CRISP system, execute the following command procedure.

```
$ @[CRISP]USER_START_DGT
```

DATAGATE will start automatically when CRISP is started. Use the following command to start CRISP and DATAGATE.

```
$ CRSTART
```

To start DATAGATE on a non-CRISP system, execute the following command procedure.

```
$ @[TAG.DGT]USER_START_DGT
```

This command line may also be inserted in your SYSTARTUP_V5.COM file so that DATAGATE will start automatically on system boot.

If you have chosen to run DATAGATE on a batch queue, a log file is created in [TAG.DGT]SUBMIT_DGT_nn.LOG, where DGT_nn is the name of the associated DATAGATE process.

Stopping DATAGATE

To stop all DATAGATE processes, execute the following command.

```
$ DGTK/PRODUCT=DGT
```

The previous line should be added to the system shutdown procedure SYS$MANAGER:SYSHUTDWN.COM prior to any lines that shut down relational database monitors.

Configuring Datagate

General

The DATAGATE User's Guide provides the information concerning the configuration of the DATAGATE software.

This section contains the following subsections.

Section	**Description**
Using DATAGATE *(page 13)*	This section of the manual contains information concerning the uses of DATAGATE.
SQL Statements *(page 14)*	This section of the manual contains information concerning DATAGATE uses of SQL, including writing to and reading from a relational database.
UCF Design *(page 16)*	This section of the manual contains information concerning the contents of the User Configuration File (UCF).
DATAGATE Triggers *(page 21)*	This section of the manual defines the use of triggers with DATAGATE.
CRISP Variable Syntax *(page 23)*	This section of the manual defines the use of CRISP variables with DATAGATE.
DATAGATE and SQL Statements *(page 24)*	This section of the manual defines how DATAGATE locates and uses SQL statements.
NULL Data *(page 35)*	This section of the manual defines how DATAGATE handles null data in relational databases.
Supported Data Types *(page 36)*	This section of the manual defines the CRISP and RDBMS data types that will be supported by DATAGATE.

(Continued on next page.)

Section	Description
DATAGATE DATE Functions *(page 37)*	This section of the manual defines how DATAGATE timestamps records that are written to the relational database.
Using CRISP Arrays *(page 38)*	This section of the manual defines how DATAGATE uses CRISP arrays.
Using DATAGATE to Submit Batch Jobs *(page 41)*	This section of the manual defines how DATAGATE can be used to submit command procedures to a batch queue.
Improving CPU Usage *(page 42)*	This section of the manual defines how to improve CPU usage.
A Sample UCF for Rdb/VMS *(page 43)*	This section of the manual contains a sample UCF for Rdb/VMS.

Using DATAGATE

DATAGATE provides users with a flexible means to transfer data both into and out of a Relational Database Management System (RDBMS). Data transfer may be accomplished between the relational database and the CRISP Application Database (ADB) in two manners.

First, users may write programs in languages such as "C" or Fortran, embedding SQL statements in the code, and then reading or writing CRISP variables by making calls to a communications library provided with DATAGATE. These programs are then under user control as opposed to CRISP control; they can be executed (via mechanisms provided with some RDBMS screen packages) when data changes or when some other RDBMS event occurs.

Second, users may preconfigure transfers by creating a file that contains a combination of DATAGATE commands and the ANSI standard Structured Query Language (SQL). Instead of placing values into the SQL statements, users will place the names of CRISP variables whose values will be substituted in the SQL statement when the statement is executed. Actual execution of a SQL statement will be determined by "triggers" from CRISP application database variables. The application database may be under the control of a CRISP logic if the user so desires.

CRISP Communications Package

DATAGATE provides a run-time library containing routines that make it simple for users to read and write CRISP variable data. Using routines in the library, users will be able to create lists (Data Source Lists (DSL)) of CRISP variables. Variables in a single list may reside on any CRISP node in the system. Once a DSL is created, a separate call is made to either read the list or write the list.

Examples of programs which contain embedded SQL and make calls to the WORF communications RTL are located in the [TAG.DGT] directory. The files that will build an Rdb/VMS demo using a VAX C compiler are as follows.

- DGT_BUILD_RDBVMS_C_EXAMPLE.COM
- DGT_C_EXAMPLE.TEMPLATE
- DGTTST.C32

Building the demo requires the following two steps.

1 If you plan to actually execute the demo, copy DGTTST.C32 to a CRISP VAX. Then, use LGBUILD to build it and LGCONFIG to install it. You will need to stop and restart CRISP.

(Continued on next page.)

CRISP Communications Package (cont)

2 Execute the DGT_BUILD_RDBVMS_C_EXAMPLE.COM procedure. This procedure builds a test Rdb/VMS database and prompts you for the name of the CRISP node where you installed the DBTTST database. When this procedure finishes executing, a DGT_RDBVMS_C_EXAMPLE.EXE will be created.

Once the DGT_FRBVMS_C_EXAMPLE.EXE file is created, it may be run. The file DGT_RDBVMS_C_EXAMPLE.SC is the SQL module containing `EXEC SQL` statements. The file DGT_RDBVMS_C_EXAMPLE.C contains the C code generated.

Similar procedures are available in [TAG.DGT] to build an Ingres demo.

User Configuration File (UCF)

When the exact relational tables involved in the data transfer are known, users may place the SQL statements describing the transfer in a UCF (User Configuration File). To allow for adhoc data transfer, users many also specify a CRISP STRING variable which will contain the SQL statement to be executed.

The user generated User Configuration File (UCF) will contain control parameters referred to as opcodes. These opcodes are converted into an optimized internal format at start time. The opcodes are used by the various processes that make up DATAGATE to control the number of logical transactions and to determine when SQL statements are to be executed. The DATAGATE UCF is an ASCII text file that is easily created with any text file editor. A sample file may be found in the [TAG.DGT] directory in the file DGT_SAMPLE.UCF.

SQL Statements

DATAGATE uses ANSI Standard SQL as its base language for communicating to an RDBMS. Depending on the RDBMS, some non-ANSI SQL features may also be available. All Data Manipulation Language (DML) statements are allowed (DELETE, INSERT, SELECT, and UPDATE) as well as all Data Definition Language (DDL) statements (ALTER, COMMENT, CREATE, and DROP). In addition, the Data Control Language (DCL) statements (GRANT and REVOKE). The COMMIT and ROLLBACK statements are also under user control.

SQL Statements (cont)

Writing To A Relational Database

In the relational database environment, there are 3 types of statements which add or change data in a database: INSERT, UPDATE and DELETE. The INSERT statement is used when the user wants to create a new row in a table. Effectively, this is an ADD function. UPDATE is used to MODIFY one or more rows in a single relational database table, while DELETE is used to delete one or more rows in a single database table.

However, any data in the relational database is not actually changed until a COMMIT is issued. The COMMIT statement means that the changes should be made permanent. A user may also issue a ROLLBACK statement which means that the changes should not be made (i.e., the database should be put back the way it was before).

Reading From A Relational Database

In the relational database environment, the SELECT statement is used to "locate" and "read" data from the relational database. A single row SELECT is fairly straight forward. As long as the user is familiar enough with the relational database to know that a single row is to be retrieved, the statement can be easily executed and the data moved into the corresponding CRISP database variables.

When a SQL SELECT statement retrieves more data than the CRISP application database has been configured for, the CRISP application can be designed to handle the data in several ways. Users may, for example, wish to use the CRISP database to "scroll" through rows in a relational database table, or combination of tables. By using a combination of the DATAGATE ROW_COUNT> function and the DATAGATE ROW_START> opcode, the user will be able to issue SQL statements that get "the first 5 rows", then the "next 5 rows", etc, until all data in the query has been transferred through the CRISP database. The order that the data is returned to the CRISP database is determined by the ORDER BY clause in the SQL statement itself. Any SQL SELECT statement is allowed, including multi-table JOINS.

Another consideration when reading data from a relational database is when to issue the ROLLBACK or COMMIT statements. Due to the transaction capabilities of all Relational Database Management Systems, once a read query has been started, the data returned by the RDBMS is guaranteed not to change until a ROLLBACK or COMMIT is issued; at that time, any data that was written by "other sources" will be made available if the query is re-executed. When reading data from the relational database, the affect of a COMMIT or ROLLBACK is to finish the query.

SQL Statements (cont)

A DATAGATE Logical Transaction

In RDBMS terms, a transaction consists of all statements that take place between the first of many DML statements and the execution of either a COMMIT or a ROLLBACK. When a user starts a transaction, his view of the database is logically frozen; any data he adds, modifies, or deletes is only changed in his view; any data he reads is only read from his view. Transactions may be read (where the SELECT is the only statement issued), or write (where any statement except the SELECT is issued), or read/write. The affect of a COMMIT on a write transaction is that any changes made during the transaction are made permanent; they will be seen by other users. The affect of a ROLLBACK on a write transaction is that the database reverts back to the way it was prior to the start of the transactions; all changes are discarded. The affect of a COMMIT or ROLLBACK on a read transaction is the same: any changes made by other users during the course of the transactions are now available, and will be seen if the SELECT is re-executed.

This discussion shows that the question of when to issue a COMMIT or ROLLBACK is important, especially when a UCF has SQL INSERT statements followed by SQL SELECT statements. DATAGATE may be in the process of returning a multiple row select by groups of rows when a COMMIT from an INSERT statement needs to be issued. If DATAGATE were to go ahead and issue the COMMIT, it is possible that the data in the SELECT statement would be changed, since new data could potentially be present.

To handle this situation, DATAGATE provides a concept known as a LOGICAL TRANSACTION. By grouping SQL statements together, the user will have the capability to determine which statements will be affected when a COMMIT or ROLLBACK is issued.

All UCFs must have at least one logical transaction (denoted by the LOGICAL_TRANSACTION> opcode). In many user scenarios, one logical transaction is sufficient. For those RDBMSs that support multiple databases, each database must have its own logical transaction. DATAGATE will create one process on the RDBMS CPU for each logical transaction found in the UCF.

UCF Design

The User Configuration File (UCF) is an ASCII text file composed of a sequential listing of commands (opcodes) for DATAGATE. It is the user's responsibility to ensure the proper organization of these opcodes for his specific purpose.

(Continued on next page.)

UCF Design (cont)

An opcode may begin at any location on the line as long as it is the first text item and is immediately followed by one or two right angle brackets. Most opcodes, with a few exceptions, allow up to two parameters. The first parameter will generally be an Application Database (ADB) variable name and the second parameter will usually be a default value for the operation. Constant (token) definitions are supplied for default entries, where appropriate, in an attempt to reduce confusion.

The general syntax of all DATAGATE opcode statement is as follows.

```
OPCODE> DBNODE::DB_NAME:VARIABLE ; DEFAULT_VALUE
OPCODE>> DBNODE:DB_NAME:VARIABLE ; DEFAULT_VALUE ! COMMENT
```

Where OPCODE is one of the opcodes described in this manual; DBNODE is the VMS node where the application database resides, always followed by a double colon ("::"); DB_NAME is the optional database identifier, always followed by a colon (":"); NAME is the name of the usually optional database variable used by the command, and DEFAULT_VALUE is the initial value or the only value used if the database variable is missing. DEFAULT_VALUE may be a literal numeric (integer or float), a pre-defined constant (token), or a quoted text-string, depending on the specific opcode used. See the example at the end of this manual for specifics.

The single ">" indicates a continuously scanned statement, while ">>" indicates a statement that is executed only once at startup or restart.

The exclamation point denotes the start of a comment. Anything to the right of the exclamation point is ignored. Imbedded comments are illegal and you may therefore start comments only to the right of any opcodes you wish to use. It is legal to have a line which contains only a comment. Further, all spaces and tabs, other than those enclosed in double quotes, as well as blank lines, will be ignored.

Many opcodes have their own system defined default value. Usually the value need not be specified if the default value matches your system requirements. Any default entry supplied to the right of the semicolon is substituted for the system default value for the current opcode only. This means that subsequent uses of the same opcode will still have the original system default value.

UCF Design

The DATAGATE Opcodes

The "control" opcodes are those that break the UCF into sections. They are as follows.

- DEFINE>
- INIT>
- PROCESS>
- LOGICAL_TRANSACTION>
- END>

The first opcode in every UCF must be the DEFINE> opcode. This opcode flags the start of the definition section. Any opcodes prior to the DEFINE> opcode are ignored. Until either the INIT> or the PROCESS> opcode is encountered, all statements following the DEFINE> opcode are considered to be initial definitions, and are processed only once at startup, or at restart. Any opcodes not considered legal in the definition section will be ignored. All opcodes in the definition section are "one shot" opcodes. This means that they will be treated as though they had two right angle brackets immediately following them. The purpose of the definition section is to define customer licensing information to the DATAGATE product. This means that each of the following opcodes must appear in the definition section:

- CUSTOMER_CONFIG_CODE>
- CUSTOMER_CONFIG_ID>
- CUSTOMER_LOCATION>
- CUSTOMER_NAME>
- CUSTOMER_SW_LICENSE>
- CUSTOMER_USE_LIMIT>

The INIT> opcode, which may optionally be declared immediately preceding the process section, indicates the beginning of the initialization section. Opcodes found in this section are considered to be initialization opcodes, and are processed only once at startup, or at restart. This section permits the user to define some opcodes that may otherwise be repeated throughout the UCF. The opcodes defined in the initialization section will become, in effect, the new defaults for that opcode. This new default maybe overridden in a corresponding LOGICAL_TRANSACTION> section.

The following opcodes (except for EXIT_IF>) may appear in the initialization section to define values on a UCF wide basis. In addition, each opcode may appear in the process section following a LOGICAL_TRANSACTION> opcode to define values for that particular logical transaction.

(Continued on next page.)

UCF Design

The DATAGATE Opcodes (cont)

- ADB_IDENT>
- ADB_NODE>
- ADB_TYPE>
- BATCH_FILE>
- BATCH_PARAMETER_P1>
- BATCH_PARAMETER_P2>
- BATCH_PARAMETER_P3>

  .
  .
  .
- BATCH_PARAMETER_P8>
- BATCH_QUE>
- BIT_NULL_DATA_VALUE>
- DISABLE_RDBMS_COMMAND>
- ENABLE_RDBMS_COMMAND>
- EXIT_IF>
- FLOAT_NULL_DATA_VALUE>
- NULLS_CHECKING>
- RDBMS_IDENT>
- RDBMS_TYPE>
- SET_MIN_TRANSACTION_TIME_TO>
- SET_VAR_TO>
- STRING_NULL_DATA_VALUE>
- TIMEOUT>
- TIMEOUT_ACTION>
- ZERO_DATE>

The PROCESS> opcode, which flags the end of the definition and initialization sections and the beginning of the process section, is also required. Any opcodes not considered legal in the process section is ignored. All logical transactions should be defined in this section. All of the following opcodes must appear in this section.

- ADD_TO_VAR>
- CLEAR_STATUS_IF>
- COMMIT_AND_CLR_IF>
- COMMIT_IF>
- DISPLAY_END_OF_DATA_AT>
- DISPLAY_MORE_DATA_AT>
- DISPLAY_NEXT_ROW_INDEX_AT>
- DISPLAY_RDBMS_STATUS_AT>
- DISPLAY_SQLCODE_AT>
- DISPLAY_SQL_DONE_AT>
- DISPLAY_TOTAL_ROWS_AT>
- EXECUTE_AND_CLR_IF>
- EXECUTE_COMMIT_AND_CLR_IF>

(Continued on next page.)

UCF Design

The DATAGATE Opcodes (cont)

- EXECUTE_COMMIT_IF>
- EXECUTE_IF>
- LOGICAL_TRANSACTION>
- PRIMARY_STATUS_AT>
- ROLLBACK_AND_CLR_IF>
- ROLLBACK_IF>
- ROW_COUNT>
- ROW_START>
- SECONDARY_STATUS_AT>
- SQL_STATEMENT>
- SUBMIT_BATCH_AND_CLR_IF>

The END> opcode, which is required, signals the end of the UCF. All opcodes following the END> opcode are ignored.

Summary:

1) All opcodes end in ">" or ">>".

2) Not all opcodes support both parts of NAME; DEFAULT_VALUE; the ";" is required when either, or both are given.

3) All licensing information must follow DEFINE> and precede PROCESS>.

4) All initialization opcodes follow INIT> and precede PROCESS>.

5) Any text following an "!" on a line will be ignored.

6) A LOGICAL_TRANSACTION> opcode must be the first following the PROCESS> opcode.

UCF Summary

To reiterate, the general format of all DATAGATE configuration statements in a UCF file is as follows.

```
COMMAND> NAME;DEFAULT_VALUE !  COMMENT
```

Where COMMAND is one of the configuration commands (Opcodes) described in this manual; NAME is the name of an usually optional database variable used by the command and DEFAULT_VALUE is the initial value or the only value used if the database variable is missing. If the database NAME is provided, the DEFAULT_VALUE will be replaced at run time by the value of the variable.

(Continued on next page.)

UCF Design

UCF Summary (cont)

A single ">" indicates a continuously scanned statement, while ">>" indicates a statement that is executed only once at startup or restart.

+ **Note:**

You must include the six customer identification commands exactly as they appear in your DATAGATE License Agreement. The commands should be placed immediately after the DEFINE> statement.

An example of a user configuration file (UCF) is located at the back of this manual.

DATAGATE Triggers

In DATAGATE, a SQL statement is executed when a trigger is TRUE. A trigger is any CRISP variable of type INTERMEDIATE, NUMERIC, LONG, or FLOAT. The trigger is considered to be TRUE if its value is non-zero; the trigger is FALSE if its value is zero. A trigger may not be of type COUNTER, TIMER, or STRING.

A CRISP variable becomes the trigger for a SQL statement when it is used with one of the following opcodes:

- EXECUTE_IF>
- EXECUTE_AND_CLR_IF>
- EXECUTE_COMMIT_IF>
- EXECUTE_COMMIT_AND_CLR_IF>

During each scan of the UCF, all triggers are read. When they become TRUE, actions are performed according to the EXECUTE type opcode as follows:

- EXECUTE_IF>
  1) Read all values necessary to complete the SQL statement.
  2) Execute the SQL statement.
  3) For SELECT statements, write the values found back to CRISP.
  4) Write any associated values found in the DISPLAY type opcodes.
- EXECUTE_AND_CLR_IF>
  1) Read all values necessary to complete the SQL statement.
  2) Execute the SQL statement.

(Continued on next page.)

DATAGATE Triggers (cont)

3) For SELECT statements, write the values found back to CRISP.

4) Write any associated values found in the DISPLAY type opcodes.

5) Write a value of 0 to the trigger variable.

- EXECUTE_COMMIT_IF>

  1) Read all values necessary to complete the SQL statement.

  2) Execute the SQL statement.

  3) For SELECT statements, write the values found back to CRISP.

  4) Write any associated values found in the DISPLAY type opcodes.

  5) Issue a COMMIT.

- EXECUTE_COMMIT_AND_CLR_IF>

  1) Read all values necessary to complete the SQL statement.

  2) Execute the SQL statement.

  3) For SELECT statements, write the values found back to CRISP.

  4) Write any associated values found in the DISPLAY type opcodes.

  5) Issue a COMMIT.

  6) Write a value of 0 to the trigger variable.

If an error is found while executing the SQL statement, only steps 1) and 2) will be performed. The only values to be written back to the CRISP database will be the SQLCODE value and the RDBMS_STATUS value if the user specified one of both of these values to be returned. In the event of an error, a COMMIT will not be issued and the trigger will not be cleared. Instead, a message is displayed either on the system console or in the log file a batch job.

CRISP Variable Syntax

CRISP variables will appear throughout the UCF. The complete syntax for a CRISP variable is as follows.

```
[node-1[,node-2]::][db:]varname[(subscript)][*]
```

Where:

node-1	is the node name of a machine that has the data.
node-2	is an alternate node name of a machine that has the data.
db	is the name of the CRISP database.
varname	is the name of the variable in the CRISP database.
subscript	is an integer, or symbol name in the database, that is used as a subscript value.
*	is a symbol used to indicate the value of the variable is to be used only once when multiple rows are to be inserted.

An example of a full variable name is as follows.

```
AZVAX1::TAGTST:WL11000(RL00000)
```

Users may use the full variable specification each time, or else use the ADB_NODE> and ADB_IDENT> opcodes to set up defaults for all CRISP variables found until the defaults are explicitly over-ridden by using a full variable specification, or by using the ADB_NODE> and/or ADB_IDENT> opcodes again to set up new defaults. Given the following excerpt from a UCF.

```
ADB_NODE> ; "AZVAX1"
ADB_IDENT> ; "TAGTST"
.*
.*
EXECUTE_IF> GET_IF113 ;
.*
.*
EXECUTE_IF> AZVAX2::OTHRDB:GET_IF111 ;
.*
.*
EXECUTE_IF> GET_IF112 ;
ADB_IDENT> ; "CSPTST"
.*
.*
EXECUTE_IF> GET_CRISP;
```

(Continued on next page.)

CRISP Variable Syntax (cont)

The variable GET_IF113 is assumed to be in the TAGTST database on node AZVAX1. GET_IF111 is found on node AZVAX2 in database OTHRDB but the defaults are still AZVAX1 and TAGTST. This means that GET_IF112 is to be found on AZVAX1 in TAGTST. Now, the ADB_IDENT> opcode is used to change the default database, but not the node; GET_CRISP should be found on node AZVAX1 in database CSPTST.

+ **Note:**

The ADB_NODE> and ADB_IDENT> opcodes MUST be used before the first CRISP variable that is not specified using the complete syntax. There is NO system default for node or database.

DATAGATE And SQL Statements

DATAGATE will find its SQL statements via the SQL_STATEMENT> opcode. DATAGATE does not process the SQL_STATEMENT opcodes until the associated trigger is true. At that time, it builds the SQL statement, parses it, and executes it. SQL statements tend to become long rather quickly, so DATAGATE will concatenate all strings found in consecutive SQL_STATEMENT> opcodes until a terminating semi-colon is found. The "trigger" for the SQL statement is found with one of the following opcodes: EXECUTE_IF>, EXECUTE_AND_CLR_IF>, EXECUTE_COMMIT_IF>, or EXECUTE_COMMIT_AND_CLR_IF>.

By default INSERT and SELECT are the only RDBMS commands available unless the user specifically enables them.

This is to prevent the user from unintentionally making disastrous changes to the database. To enable a specific command, use the ENABLE_RDBMS_COMMAND> opcode. The tokens of ALTER, COMMENT, CREATE, DELETE, DROP, GRANT, INSERT, REVOKE, SELECT, and UPDATE have been provided to make it easy to specify the command to be enabled. These are the only commands that are currently supported by DATAGATE in a SQL statement. It is also possible to disable a command by using the DISABLE_RDBMS_COMMAND> opcode.

Note that the SQL commands COMMIT and ROLLBACK are missing from the list of allowable commands. This is because DATAGATE provides the means to do this through special opcodes other than the SQL_STATEMENT> opcode. When setting the trigger for a SQL statement, a COMMIT is implied if either the EXECUTE_COMMIT_IF> or EXECUTE_COMMIT_AND_CLR_IF> opcodes are used. A COMMIT will be issued if the trigger associated with either the COMMIT_IF> or COMMIT_AND_CLR_IF> opcodes is true. There are no opcodes that execute a SQL statement and then issue a ROLLBACK; it is assumed that most data is expected to be made permanent in the relational database. It is expected that a ROLLBACK will only be issued in the event of an error.

(Continued on next page.)

DATAGATE and SQL Statements (cont)

If a CRISP application logic is in control of the DATAGATE data transfers, make sure that either the DISPLAY_SQLCODE_AT> opcode or the DISPLAY_RDBMS_STATUS> opcode is used with each SQL statement. DATAGATE will not clear the trigger variable unless the SQLCODE status returned by the RDBMS is non-negative; the SQLCODE status will be written back to the CRISP database even in the event of an error. Refer to the following sections for examples.

DATAGATE will locate the names of CRISP variables in the SQL statements by the pound sign (#) which must precede them. In the following SQL statement, the CRISP variables are EMP_VAR and LAST_VAR.

```
SQL_STATEMENT> ; "INSERT INTO EMPLOYEES"
SQL_STATEMENT> ; "(EMPLOYEE_ID, LAST_NAME)"
SQL_STATEMENT> ; "VALUES"
SQL_STATEMENT> ; "(#EMP_VAR, #LAST_VAR);"
```

If DATAGATE executed the previous SQL statement, the values of EMP_VAR and LAST_VAR would be substituted before the statement is executed.

+ **Note:**

Each SQL statement may be made up from many SQL_STATEMENT> opcodes but must have a single execute type opcode. The variables associated with the execute opcodes for different SQL statements may be the same so that several SQL statements will execute when a single trigger is set. In this case, the statements will be executed in the order they are found in the UCF. This is referred to as 'chaining' SQL statements. It is suggested that the EXECUTE_IF> opcode be used for all SQL statements except the last one; the opcode for the final chained SQL statement should be EXECUTE_AND_CLR_IF> or EXECUTE_COMMIT_AND_CLR_IF>.

How To Do A SQL INSERT

Users will specify the INSERT statement when they want to add one or more rows to a relational database table. DATAGATE does no checking to verify that the relation specified is in fact a table and not a view. Inserting a row into a view may cause the row to be "lost". (For more information on views see your RDBMS documentation).

The following is an example from a UCF file.

```
ADB_NODE>                            ; "AZVAX2"
ADB_IDENT>                           ; "EMPADB"
SQL_STATEMENT> ; "INSERT INTO EMPLOYEES"
SQL_STATEMENT> ; "(EMPLOYEE_ID, LAST_NAME,FIRST_NAME)"
SQL_STATEMENT> ; "VALUES"
SQL_STATEMENT> ; "(#EMPNO_VAR, #LAST_VAR,#FIRST_VAR);"
DISPLAY_SQLCODE_AT>       STATUS_VAR ;
DISPLAY_SQL_DONE_AT>  INSERT_DONE_VAR ;
EXECUTE_COMMIT_AND_CLR_IF> INSERT_VAR ;
```

(Continued on next page.)

DATAGATE and SQL Statements

How To Do A SQL INSERT (cont)

When the value of INSERT_VAR (in CRISP application database EMPADB on VAX node AZVAX2) is TRUE, DATAGATE performs the following:

1) Read the variables EMPNO_VAR, LAST_VAR, FIRST_VAR from the CRISP database named EMPADB on node AZVAX2.

2) Substitute the values read into the SQL statement.

"INSERT INTO EMPLOYEES (EMPLOYEE__ID,.LAST_NAME, FIRST_NAME, …);"

3) If no errors are found, issue a COMMIT.

4) Write the RDBMS SQLCODE value to the variable STATUS_VAR.

5) If no errors were found, write a value of 0 to the variable INSERT_VAR and a value of 1 to the variable INSERT_DONE_VAR.

How To Do A SQL UPDATE

Users will specify the UPDATE statement when they want to modify one or more rows in a relational database table. DATAGATE does no checking to verify that the relation specified is in fact a table and not a view. Updating a view is not recommended. For more information on views refer to your RDBMS documentation).

UPDATEs are performed in the same fashion as INSERTs with one exeception: the UPDATE command must first be enabled. Failure to enable the UPDATE will prevent the execution of the SQL statement. In this case, an error message is then sent to the system console or written to the log file.

For purposes of the following example, it is assumed that the ADB_IDENT> opcode has already been used so that the CRISP variables (indicated by pound signs (#)) found in the SQL statements can be located in a CRISP ADB.

```
ENABLE_RDBMS_COMMAND>                                ;UPDATE
SQL_STATEMENT> ; "UPDATE EMPLOYEES"
SQL_STATEMENT> ; "SET LAST_NAME = #LAST_VAR,"
SQL_STATEMENT> ; "FIRST_NAME = #FIRST_VAR"
SQL_STATEMENT> ; "WHERE EMPLOYEE_ID = #EMPNO_VAR;"
DISPLAY_SQLCODE_AT>             UPD_STATUS_VAR ;
DISPLAY_SQL_DONE_AT>           UPDATE_DONE_VAR ;
EXECUTE_COMMIT_AND_CLR_IF>          UPDATE_VAR ;
```

(Continued on next page.)

DATAGATE and SQL Statements

How To Do A SQL UPDATE (cont)

Because UPDATEs have been enabled, whenever the value of UPDATE_VAR (in the associated CRISP ADB) is non-zero, DATAGATE will perform the following.

1) Read the variables EMPNO_VAR, LAST_VAR, FIRST_VAR from the associated CRISP database.
2) Substitute the values found into the SQL statement.

"UPDATE EMPLOYEES SET LAST=NAME = #LAST_VAR, … , #EMPNO_VAR;"

3) If no errors are found, issue a COMMIT.
4) Write the RDBMS SQLCODE value to the variable UPD_STATUS_VAR.
5) If no errors were found, write a value of 0 to the variable UPDATE_VAR and a value of 1 to the variable UPDATE_DONE_VAR.

How To Do A SQL DELETE

Users should use caution when performing the SQL DELETE function. It has been provided with the DATAGATE interface for the sake of completeness. As with the UPDATE function, the DELETE command must first be enabled before it can be used.

For purposes of the following example, it is assumed that the ADB_IDENT> opcode has already been used so that the variables associated with the pound signs (#) found in the SQL statements can be located in a CRISP ADB.

```
ENABLE_RDBMS_COMMAND>                           ; DELETE
SQL_STATEMENT> ; "DELETE FROM EMPLOYEES"
SQL_STATEMENT> ; "WHERE EMPLOYEE_ID = #EMPNO_VAR;"
DISPLAY_SQLCODE_AT>           DEL_STATUS_VAR ;
DISPLAY_SQL_DONE_AT>         DELETE_DONE_VAR ;
EXECUTE_AND_CLR_IF>               DELETE_VAR ;
DISABLE_RDBMS_COMMAND>                          ; DELETE
```

Because DELETEs have been enabled, when the value of DELETE_VAR (in the associated CRISP ADB) is non-zero, DATAGATE will perform the following.

(Continued on next page.)

DATAGATE and SQL Statements

How To Do A SQL DELETE (cont)

1) Read the variable EMPNO_VAR from the associated CRISP database.

2) Substitute the value found into the SQL statement.

"DELETE FROM EMPLOYEES WHERE EMPLOYEE__ID = #EMPNO_VAR;"

3) Write the RDBMS SQLCODE value to the variable DEL_STATUS_VAR.

4) If no errors were found, write a value of 0 to the variable DELETE_VAR and a value of 1 to the variable DELETE_DONE_VAR

How To Do A SQL SELECT

The DATAGATE form of the SQL SELECT statement is the only one that looks slightly different from standard interactive SQL. The form of the SELECT statement in DATAGATE resembles a programmatic singleton SELECT with use of the INTO clause. The variables in the INTO clause will be the CRISP variables where DATAGATE is to write the data retrieved from the SELECT statement. As with the other DATAGATE SQL syntax, these CRISP variables are to be preceded by a pound sign (#).

There is no need to specifically enable the SELECT command as it is enabled by default.

Retrieving A Single Row

The following is an example of the simplest form of the SELECT. In this example, exactly one row is retrieved from the relational database and the values written to CRISP; it is assumed that the EMPLOYEE_ID field in the database is unique and that all CRISP variables are of type STRING; it is also assumed that the ADB_IDENT> opcode has already been used so that the variables associated with the pound signs (#) found in the SQL statements can be located in a CRISP ADB.

(Continued on next page.)

DATAGATE and SQL Statements

How To Do A SQL SELECT

Retrieving A Single Row (cont)

```
STRING_NULL_DATA_VALUE>                    ; "No data"
SQL_STATEMENT> ; "SELECT LAST_NAME, FIRST_NAME"
SQL_STATEMENT> ; "INTO #LAST_VAR, #FIRST_VAR"
SQL_STATEMENT> ; "FROM EMPLOYEES"
SQL_STATEMENT> ; "WHERE EMPLOYEE_ID = #EMPNO_VAR;"
DISPLAY_SQLCODE_AT>         SEL_STATUS_VAR ;
DISPLAY_SQL_DONE_AT>       SELECT_DONE_VAR ;
EXECUTE_AND_CLR_IF>             SELECT_VAR ;
```

When the value of SELECT_VAR (in the associated CRISP ADB) is non-zero, DATAGATE will perform the following.

1) Read the variable EMPNO_VAR from the associated CRISP database.

2) Substitute the value found into the SQL statement.

"SELECT LAST_NAME, FIRST_NAME, …, #EMPNO_VAR);"

3) If a matching row is found, take the value found in the LAST_NAME column and write it to the CRISP variable LAST_VAR and take the value found in the FIRST_NAME column and write it to the CRISP variable FIRST_VAR.

4) If no matching row is found, the values associated with the STRING_NULL_DATA_VALUE> opcode will be written to the associated CRISP variables. In this case the value "No data" will be written to each of the variables LAST_VAR and FIRST_VAR.

5) Write the RDBMS SQLCODE value to the variable SEL_STATUS_VAR.

6) If no errors were found, write a value of 0 to the variable SELECT_VAR and a value of 1 to the variable SELECT_DONE_VAR

DATAGATE and SQL Statements

How To Do A SQL SELECT (cont)

Retrieving Multiple Rows

The following example shows how to read all rows in a table one at a time. It is the ROW_COUNT> opcode that specifies that only one row is to be retrieved at a time. The ROW_START> opcode specifies which row in the table is to be found and written to the CRISP ADB.

```
SET_VAR_TO>>                 ROW_START_VAR ; 1
STRING_NULL_DATA_VALUE>                    ; "No Data"
ROW_START>                   ROW_START_VAR ;
ROW_COUNT>                                 ; 1
SQL_STATEMENT> ; "SELECT LAST_NAME, FIRST_NAME"
SQL_STATEMENT> ; "INTO #LAST_VAR, #FIRST_VAR"
SQL_STATEMENT> ; "FROM EMPLOYEES"
SQL_STATEMENT> ; "ORDER BY LAST_NAME;"
DISPLAY_END_OF_DATA_AT>         COMMIT_VAR ;
DISPLAY_TOTAL_ROWS_AT>           TOTAL_VAR ;
DISPLAY_NEXT_ROW_INDEX_AT>   ROW_START_VAR ;
EXECUTE_AND_CLR_IF>             SELECT_VAR ;
COMMIT_AND_CLR_IF>              COMMIT_VAR ;
```

When DATAGATE performs its initialization, it will write a value of 1 to the variable ROW_START_VAR. It will do this only once (as shown by the double angle bracket (>>)). It is assumed for purposes of this example, that DATAGATE encounters no errors; it is also assumed that the EMPLOYEES table has only 2 rows. The ORDER BY clause in the SQL statement itself determines which of the 2 rows is "first" and which one is "second".

The following shows how the rows are retrieved the first time that SELECT_VAR is set:

1) Since the value of ROW_START_VAR is one, read the first row in the EMPLOYEES table.

2) Write the following values to the CRISP database:

   a) Take the value found in the LAST_NAME column of the first row and write it to the CRISP variable LAST_VAR.

   b) Take the value found in the FIRST_NAME column of the first row and write it to the CRISP variable FIRST_VAR.

(Continued on next page.)

DATAGATE and SQL Statements

How To Do A SQL SELECT

Retrieving Multiple Rows (cont)

c) Since the row returned was row 1, the next available row is row 2; therefore write the value of 2 to ROW_START_VAR. (This is due to the DISPLAY_NEXT_ROW_INDEX_AT> opcode.)

d) Since the last row was not found, write a value of 0 to COMMIT_VAR. (This is due to the DISPLAY_END_OF_DATA_AT> opcode.)

e) Since the first row in the table was returned, write a value of 1 to TOTAL_VAR. (This is due to the DISPLAY_TOTAL_ROWS_AT> opcode.)

3) If no errors were found, write a value of 0 to the variable SELECT_VAR.

The next time that the SELECT_VAR variable is found to be TRUE, the following will occur.

1) Since the value of ROW_START_VAR is now 2, read the second row in the EMPLOYEES table.

2) Write the following values to the CRISP database:

   a) Take the value found in the LAST_NAME column for the second row and write it to the CRISP variable LAST_VAR

   b) Take the value found in the FIRST_NAME column for the second row and write it to the CRISP variable FIRST_VAR.

   c) Since the row returned was row 2, the next available row is row 3; therefore write the value of 3 to ROW_START_VAR.

   d) Since the last row was not found, write a value of 0 to COMMIT_VAR.

   e) Since the second row in the table was returned, write a value of 2 to TOTAL_VAR.

3) If no errors were found, write a value of 0 to the variable SELECT_VAR.

(Continued on next page.)

DATAGATE and SQL Statements

How To Do A SQL SELECT

Retrieving Multiple Rows (cont)

The next time that the SELECT_VAR variable is found to be TRUE, the following will occur.

1) Since the value of ROW_START_VAR is now 3, try to read the third row in the EMPLOYEES table. It is not found.

2) Write the following values to the CRISP database:

   a) Take the value found in the STRING_NULL_DATA_VALUE> opcode ("No Data")and write it to the CRISP variable LAST_VAR.

   b) Take the value found in the STRING_NULL_DATA_VALUE> opcode ("No Data")and write it to the CRISP variable FIRST_VAR.

   c) Since a row was not returned, the next available row is row 1, therefore write the value of 1 to ROW_START_VAR.

   d) Since the last row was found, write a value of 1 to COMMIT_VAR.

   e) Since the second row in the table was the last to be returned, write a value of 2 to TOTAL_VAR.

3) If no errors were found, write a value of 0 to the variable SELECT_VAR.

4) Because COMMIT_VAR is now set, issue a COMMIT and write a value of 0 to the COMMIT_VAR variable.

DATAGATE and SQL Statements (cont)

Adhoc SQL

Adhoc SQL is the ability to take the entire SQL statement itself from CRISP variables. Users may consecutively list the SQL_STATEMENT> opcode many times in the UCF with the names of STRING variables that are to hold a SQL statement. When the trigger associated with the SQL statement is TRUE, DATAGATE will read each of the variables, concatenating them together until the last character found is a semicolon (;).

Users may also use a combination of CRISP variables and hard-coded default values which will be concatenated together when the associated trigger becomes TRUE.

As an example, suppose users wished to create a table at the end of each day. The columns in the tables would be the same, but the name of the table would be dependent on some criteria established in a CRISP application logic. Users would do the following in the UCF.

```
ENABLE_RDBMS_COMMAND>                         ; CREATE
SQL_STATEMENT> ; "CREATE TABLE"
SQL_STATEMENT>                  TABLE_NAME_VAR ;
SQL_STATEMENT> ; "( COLUMN_1   CHAR(5),
SQL_STATEMENT> ; "  COLUMN_2   REAL,
SQL_STATEMENT> ; "  COLUMN_3   LONG);"
DISPLAY_SQLCODE_AT>          CREATE_STATUS_VAR ;
DISPLAY_SQL_DONE_AT>         TABLE_CREATED_VAR ;
EXECUTE_COMMIT_AND_CLR_IF>    CREATE_TABLE_VAR ;
DISABLE_RDBMS_COMMAND>                        ; CREATE
```

While highly flexible, the previous scenario is CPU intensive. Use this feature only if speed is not an issue. Also, where possible, avoid placing these types of SQL statements in the same logical transaction as SQL statements that are time critical.

Another example of the capability to specify SQL statements in CRISP variables is the following.

```
SQL_STATEMENT> ; "SELECT LAST_NAME, FIRST_NAME"
SQL_STATEMENT> ; "INTO #LAST_VAR, #FIRST_VAR"
SQL_STATEMENT> ; "FROM EMPLOYEES"
SQL_STATEMENT> ; "ORDER BY"
SQL_STATEMENT>                         ORDER_VAR ;
SQL_STATEMENT> ; ";"
ROW_COUNT>                                       ; 1
ROW_START>                         ROW_START_VAR ;
DISPLAY_NEXT_ROW_INDEX_AT>         ROW_START_VAR ;
DISPLAY_SQLCODE_AT>            SELECT_STATUS_VAR ;
DISPLAY_SQL_DONE_AT>             SELECT_DONE_VAR ;
DISPLAY_END_OF_DATA_AT>               COMMIT_VAR ;
EXECUTE_AND_CLR_IF>           SELECT_TRIGGER_VAR ;
COMMIT_AND_CLR_IF>                    COMMIT_VAR ;
```

(Continued on next page.)

DATAGATE and SQL Statements

Adhoc SQL (cont)

In the previous example, the name of the column to sort by has not been specified in the SQL statement. Instead, its value is to come from a CRISP variable. Depending on the value of the CRISP variable (which must be the name of a column in the EMPLOYEES table), the data will be sorted differently.

DATAGATE and SQL Statements (cont)

Retrieving Data From Multiple CRISP Databases

Users do not need to limit SQL statements that retrieve data from a single CRISP database. Simply use the ADB_NODE> and ADB_IDENT> opcodes for one database, and then use the full variable specification for any variables not in the default node/database. The following is a simple example.

```
ADB_NODE>      ; "AZVAX1"
ADB_IDENT>     ; "TAGTST"
SQL_STATEMENT> ; "INSERT INTO EMPLOYEES"
SQL_STATEMENT> ; "(EMPLOYEE_ID, LAST_NAME, FIRST_NAME)"
SQL_STATEMENT> ; "VALUES"
SQL_STATEMENT> ; "(#EMP_VAR, #CSPTST:LAST_VAR,"
SQL_STATEMENT> ; "#AZVAX2::FIRST_VAR);"
EXECUTE_IF>    AZVAX3::TSTTST:TRIGGER_VAR ;
```

The following table shows where each of the four CRISP variables are located.

Variable Name	CRISP Node	CRISP Database
EMP_VAR	AZVAX1	TAGTST
LAST_VAR	AZVAX1	CSPTST
FIRST_VAR	AZVAX2	TAGTST
TRIGGER_VAR	AZVAX3	TSTTST

NULL Data

The concept of a NULL is very important in relational database theory, however there is no similar concept in the CRISP language. A means is therefore provided to interpret NULL data in the relational database. Users will use the *_NULL_DATA_VALUE> opcodes, along with the NULLS_CHECKING> opcode so that data is correctly interpreted by DATAGATE.

First, when SELECTing data from the relational database and writing that data back to CRISP, it is possible that a value in a relational column IS NULL. This means that there is no data to write back to CRISP. DATAGATE will write the values to CRISP that are associated with the following opcodes:

CRISP Data Type	DATAGATE Opcodes	Default Value
STRING	STRING_NULL_DATA_VALUE>	""
LONG	NUMERIC_NULL_DATA_VALUE>	-99
NUMERIC	NUMERIC_NULL_DATA_VALUE>	-99
FLOAT	FLOAT_NULL_DATA_VALUE>	-99.0
LOGICAL	BIT_NULL_DATA_VALUE>	0

Users may use these defaults, or may use the opcodes to override the default values. Each of the above opcodes may be used with a CRISP variable, however only one of each of the above opcodes may appear in each logical transaction.

If users need to write NULL values to a relational database column, the NULLS_CHECKING> opcode is used. The tokens ON and OFF are provided to determine whether a NULL or a value is to be written to the relational database. If a NULL is to be written to the relational database NULLS_CHECKING must be ON and the value found in the CRISP variable must be the same as the corresponding null data value.

As an example:

```
SET_VAR_TO>> FLOAT_VAR ; -99.0
SET_VAR_TO>> NUMBER_VAR ; 2200
FLOAT_NULL_DATA_VALUE> ; -99.0
!
NULLS_CHECKING> ; ON
SQL_STATEMENT> ; "INSERT INTO TEST_TABLE"
SQL_STATEMENT> ; "(FLOAT_COL_1, NUMERIC_COL_1)"
SQL_STATEMENT> : "VALUES"
SQL_STATEMENT> ; "(#FLOAT_VAR, #NUMBER_VAR);"
EXECUTE_IF> TRIGGER_VAR ;
!
NULLS_CHECKING> ; OFF
SQL_STATEMENT> ; "INSERT INTO TEST_TABLE"
SQL_STATEMENT> ; "(FLOAT_COL_1, NUMERIC_COL_1)"
SQL_STATEMENT> : "VALUES"
SQL_STATEMENT> ; "(#FLOAT_VAR, #NUMBER_VAR);"
EXECUTE_AND_CLR_IF> TRIGGER_VAR ;
```

(Continued on next page.)

NULL Data (cont)

When TRIGGER_VAR is true, two rows will be written to the table TEST_TABLE as follows:

FLOAT_COL_1	NUMERIC_COL_1
NULL	2200
-99.0	2200

It is NOT possible to use NULLS_CHECKING to bypass the IS NULL portion of a SQL WHERE clause. If you need to retrieve rows where a column value IS NULL, the WHERE clause of the SQL statement must contain the IS NULL predicate.

Supported Data Types

The following table describes the CRISP and RDBMS data types that will be supported. It should be noted that the data types in the RDBMS column are ANSI Standard SQL keywords.

CRISP	RDBMS	SIZE
NUMERIC	SMALLINT	2 BYTE
LONG	INTEGER	4 BYTE
INTERMEDIATE	SMALLINT	2 BYTE
FLOAT	REAL	4 BYTE
STRING	CHAR(n)	n BYTE

DATAGATE will automatically perform numeric data type conversions (i.e., CRISP FLOAT to RDBMS INTEGER or RDBMS SMALLINT to CRISP LONG, as necessary). DATAGATE does not support character to numeric or numeric to character conversions.

In addition, DATAGATE supports data transfers in and out of the RDBMS DATE data type as follows.

From	To
CRISP LONG	RDBMS DATE
CRISP STRING	RDBMS DATE
RDBMS DATE	CRISP LONG
RDBMS DATE	CRISP STRING

When DATAGATE encounters a transfer of data from an RDBMS DATE type to a CRISP STRING type, it will write the date to the string in the standard VMS character date format:

```
DD-MMM-YYYY HH:MM:SS.SS
```

Similarly, when DATAGATE performs a CRISP STRING to RDBMS DATE transfer, the CRISP STRING must contain data in the above specified format (IE 14-JUL-1990 23:00:00.00).

(Continued on next page.)

Supported Data Types (cont)

When DATAGATE encounters a transfer of data from an RDBMS DATE to a CRISP LONG, it will interpret the date as the number of seconds since the default zero date and write that value as an integer to the specified CRISP variable; a negative value means that the date returned by the RDBMS was prior to the default zero date; a positive value means that the date is after the default zero date. Unless users override the zero date using the ZERO_DATE> opcode, the default zero date is:

"01-APR-1970 00:00:00.00"

Refer to the description of the ZERO_DATE> opcode in the Opcode Definitions section.

Similarly, for conversions from CRISP LONG variables to RDBMS DATE columns, the data in the CRISP variable will be interpreted as the number of seconds since the zero date before it is converted to the RDBMS DATE type.

By providing the two types of DATE conversions, DATAGATE allows users the ability to display dates on CRISP workstations (using STRING variables), or to easily compare dates in the application logic (using LONG variables).

DATAGATE DATE Functions

The predefined DATE functions provide users with a simple way to timestamp records that are written to the relational database. The following two functions are provided.

- #<SYSTEM_DATE>
- #<SYSTEM_TIME>

When #<SYSTEM_TIME> is specified, DATAGATE takes the current time from the VMS clock and stores that value in the associated column. When #<SYSTEM_DATE> is specified, the hour:minute:second portion of the time is zeroed, so that the time written to the database is midnight while the date portion remains unchanged.

The following is an example of an INSERT statement using the time function.

```
SQL_STATEMENT> ; "INSERT INTO BATCH_REPORT"
SQL_STATEMENT> ; "(TIMESTAMP, BATCH_NUMBER, LOT_VALUE)"
SQL_STATEMENT> ; "VALUES ("
SQL_STATEMENT> ; "#<SYSTEM_TIME>, #BATCH_VAR,
SQL_STATEMENT> ; "#LOT_VAR);"
```

In the example, the TIMESTAMP column in the BATCH_REPORT table could be any of the RDBMS data types: DATE, CHAR, or INTEGER. DATAGATE will perform the DATE conversion for any of these types.

(Continued on next page.)

DATAGATE DATE Functions (cont)

For Rdb/VMS, the date functions may also be used as part of the WHERE
clause in a SQL SELECT, UPDATE, or DELETE statement:

```
   SQL_STATEMENT> ; "SELECT BATCH_REPORT_NUMBER,"
   SQL_STATEMENT> ; "SWITCH_VALUE"
   SQL_STATEMENT> ; "INTO #BATCH_NUMBER_VAR, #SWITCH_VAL"
   SQL_STATEMENT> ; "FROM SOME_TABLE"
   SQL_STATEMENT> ; "WHERE TIMESTAMP = #<SYSTEM_DATE>"
```

Due to limitations in the Ingres Dynamic SQL interface, DATE Ö LONG conversions and DATE functions are not allowed in the WHERE clause of an UPDATE, SELECT, or DELETE statement.

Using CRISP Arrays

By using CRISP arrays, users will be able to INSERT or SELECT more than one row at time. The ROW_COUNT> opcode is used to tell DATAGATE either how many rows to INSERT or how many rows to SELECT. The ROW_COUNT> opcode is ignored for all other types of SQL statements.

ROW_COUNT> And The SQL INSERT Statement

For INSERT statements, the ROW_COUNT> opcode tells DATAGATE how many rows are to be inserted; it also is the number of CRISP array elements that are to be used to make up those rows.

Given the following portion of a UCF.

```
   SQL_STATEMENT> ; "INSERT INTO SHIFT_AVERAGES"
   SQL_STATEMENT> ; "(TIMESTAMP, ENG_UNIT_NUM,"
   SQL_STATEMENT> ; "ENG_UNIT_VALUE)"
   SQL_STATEMENT> ; "VALUES"
   SQL_STATEMENT> ; "#<SYSTEM_TIME>, #ENG(0),"
   SQL_STATEMENT> ; "#ENGVL(0) );"
   DISPLAY_SQLCODE_AT>                  STATUS_VAR ;
   DISPLAY_SQL_DONE_AT>               SQL_DONE_VAR ;
   ROW_COUNT>                                      ; 5
   EXECUTE_COMMIT_AND_CLR_IF>          EXECUTE_VAR ;
```

When the variable EXECUTE_VAR is TRUE, DATAGATE will perform the following:

1) Read the CRISP variables so that 5 rows are inserted into the specified table as follows:

	TIMESTAMP	ENG_UNIT	ENG_UNIT_VALUE
Row 1:	INSERT Time	Value of ENG(0)	Value of ENGVL(0)
Row 2:	INSERT Time	Value of ENG(1)	Value of ENGVL(1)
Row 3:	INSERT Time	Value of ENG(2)	Value of ENGVL(2)
Row 4:	INSERT Time	Value of ENG(3)	Value of ENGVL(3)
Row 5:	INSERT Time	Value of ENG(4)	Value of ENGVL(4)

(Continued on next page.)

Using CRISP Arrays

ROW_COUNT> And The SQL INSERT Statement (cont)

2) If no errors are found, issue a COMMIT.

3) Write the RDBMS SQLCODE value (which will be 0 if there are no errors) to the STATUS_VAR variable.

4) If no errors are found, write a value of 1 to SQL_DONE_VAR and a value of 0 to EXECUTE_VAR.

In this particular example, the TIMESTAMP column values are not guaranteed to be the same; the time value itself is determined at the exact moment that DATAGATE builds the row. See the section titled "Combining Arrays and Single Values" for information on how to insert multiple rows with a single <SYSTEM_TIME> value.

ROW_COUNT>, ROW_START> And The SQL SELECT Statement

DATAGATE users have the ability to bring in multiple rows from a query by using the ROW_COUNT> opcode to indicate how many contiguous CRISP database variables are available to hold data, and then to specify the first of these variables in the SQL statement. The ROW_START> opcode is used to indicated to DATAGATE which row in the query is to be returned as the first row to the CRISP database. By using the DISPLAY_NEXT_ROW_INDEX_AT> opcode in conjunction with ROW_START>, this process is easily maintained.

Here is an example from a UCF file.

```
SQL_STATEMENT> ; "SELECT ENG_UNIT_NUM, ENG_UNIT_VALUE"
SQL_STATEMENT> ; "INTO #ENGU(0), #ENGVAL(4)"
SQL_STATEMENT> ; "FROM SHIFT_AVERAGES"
SQL_STATEMENT> ; "WHERE ENG_UNIT_NUM"
SQL_STATEMENT> ; "= #ENGUVAL"
SQL_STATEMENT> ; "ORDER BY ENG_UNIT_VALUE;"
ROW_START>                      ROW_START_VAR ;
ROW_COUNT>                                    ; 5
DISPLAY_NEXT_ROW_INDEX_AT>      ROW_START_VAR ;
DISPLAY_END_OF_DATA_AT>            COMMIT_VAR ;
EXECUTE_AND_CLR_IF>               EXECUTE_VAR ;
COMMIT_IF>                         COMMIT_VAR ;
```

Assume for purposes of this example that the value of CRISP variable ENGUVAL is 10, and that table SHIFT_AVERAGES has 9 rows where the value of column ENG_UNIT_NUM is 10. Also assume that ROW_START_VAR has a value of 1 when EXECUTE_VAR is set and that no RDBMS errors are found. The first time that EXECUTE_VAR is set, the following takes place.

(Continued on next page.)

Using CRISP Arrays

ROW_COUNT>, ROW_START> And The SQL SELECT Statement (cont)

1) The value of ENGUVAL will be read from CRISP and placed in the SQL statement.

2) Five rows (the ROW_COUNT> value) will be returned to CRISP as follows.

   Values from Row 1 are written to: ENGU(0) & ENGVAL(4)
   Values from Row 2 are written to: ENGU(1) & ENGVAL(5)
   Values from Row 3 are written to: ENGU(2) & ENGVAL(6)
   Values from Row 4 are written to: ENGU(3) & ENGVAL(7)
   Values from Row 5 are written to: ENGU(4) & ENGVAL(8)

3) A value of 6 will be written to ROW_START_VAR; a value of 0 is written to COMMIT_VAR.

4) The trigger is cleared (a value of 0 is written to EXECUTE_VAR).

Assuming that no changes are made to any data from an application logic, the next time EXECUTE_VAR is set, the following takes place.

1) Five rows (the ROW_COUNT> value) will be returned to CRISP as follows.

   Values from Row 6 are written to: ENGU(0) & ENGVAL(4)
   Values from Row 7 are written to: ENGU(1) & ENGVAL(5)
   Values from Row 8 are written to: ENGU(2) & ENGVAL(6)
   Values from Row 9 are written to: ENGU(3) & ENGVAL(7)
   Null data values are returned to: ENGU(4) & ENGVAL(8)

2) A value of 1 will be written to ROW_START_VAR (since all rows have been returned); a value of 1 is written to COMMIT_VAR.

3) The trigger is cleared (a value of 0 is written to EXECUTE_VAR).

4) Since COMMIT_VAR is now set, a COMMIT is issued.

The next time that EXECUTE_VAR is set, the query will start over.

+ **Note:**

Multiple row SELECTs and INSERTs are only available into and out of CRISP arrays.

Using CRISP Arrays (cont)

Combining Arrays And Single Values

Users can use this feature to insert multiple rows in a relational database table using the ROW_COUNT> opcode, where some values are to come from arrays, and other values are to be fixed for each row inserted. This method may also be used to assure that all rows inserted at the same time have the same <SYSTEM_TIME> or <SYSTEM_DATE> value.

The manner in which DATAGATE distinguishes which values are to remain fixed is by the appearance of an asterisk (*) following the last character of the CRISP variable name. Given the following excerpt from a UCF.

```
SQL_STATEMENT> ; "INSERT INTO SHIFT_AVERAGES ("
SQL_STATEMENT> ; "TIMESTAMP, ENG_UNIT_NUM, ENG_UNIT_TYPE,"
SQL_STATEMENT> ; "ENG_UNIT_VALUE) VALUES ("
SQL_STATEMENT> ; "#<SYSTEM_TIME>*, #ENGU(0)*, #ENGT*,"
SQL_STATEMENT> ; "#ENG_VAL(0) );"
ROW_COUNT>                                    ; 5
EXECUTE_COMMIT_AND_CLR_IF>         EXECUTE_VAR ;
```

Because the ROW_COUNT> value is 5, DATAGATE will INSERT 5 rows into the SHIFT_AVERAGES table when EXECUTE_VAR is TRUE. The rows inserted will be as follows:

```
       TIMESTAMP           ENG_UNIT_NUM       ENG_UNIT_VALUE
Row 1: Insert time Row 1  Value of ENGU(0)  Value of ENG_VAL(0)
Row 2: Insert time Row 1  Value of ENGU(0)  Value of ENG_VAL(1)
Row 3: Insert time Row 1  Value of ENGU(0)  Value of ENG_VAL(2)
Row 4: Insert time Row 1  Value of ENGU(0)  Value of ENG_VAL(3)
Row 5: Insert time Row 1  Value of ENGU(0)  Value of ENG_VAL(4)
```

The asterisk (*) following the <SYSTEM_TIME> function shows that all rows are to have the same time; the asterisk (*) following ENGU(0) shows that this value is also to remain fixed, as does the asterisk following ENGT.

Using DATAGATE To Submit Batch Jobs

Users will be able to write command procedures that will be submitted
to a specified batch queue when a CRISP variable is TRUE. This functionality has been provided to allow users to execute procedures on the RDBMS machine that are triggered by events in CRISP.

The BATCH_FILE> opcode is used to specify the name of the command procedure that is to be submitted. The value must be a full file specification including device and directory. The file type must be .COM. There is no default batch file.

(Continued on next page.)

Using DATAGATE To Submit Batch Jobs (cont)

The BATCH_QUE> opcode is used to specify the name of the batch queue that DATAGATE is to use when submitting the file specified in BATCH_FILE>. The default batch queue is SYS$BATCH.

The BATCH_PARAMETER_P*> opcodes are provided to allow users to pass parameters P1 through P8 to the command procedure. BATCH_PARAMETER_P1 is used to specify P1, BATCH_PARAMETER_P2 is used to specify P2, etc. Users may specify the value as an default quoted string, or a CRISP string variable may be used. If a parameter is not specified, its value is " ".

The SUBMIT_BATCH_AND_CLR_IF> opcode specifies the name of the crisp variable that is to be used as a trigger. When the value of the variable is true, DATAGATE submits the file specified in the BATCH_FILE> opcode to the queue specified in the BATCH_QUE> opcode with the parameters specified in the optional BATCH_PARAMETER_P* opcodes. In addition, a log file with the same directory and file name as the .COM file are created with an extension of .LOG.

```
BATCH_FILE>   ; "DISK$USER:[RDB.REPORTS]END_OF_BATCH.COM"
BATCH_QUE>    ; "SYS$BATCH"
BATCH_PARAMETER_P1>  ;  "Y"
BATCH_PARAMETER_P2>  ;  "NONE"
BATCH_PARAMETER_P3>  ;  P3_STRING_VAR ;
SUBMIT_BATCH_AND_CLR_IF> REPORTS_VAR ;
```

Improving CPU Usage

The SET_MIN_TRANSACTION_TIME_TO> opcode determines the minimum time the opcodes in a logical transaction are processed. This is the amount of time it takes to read all triggers, determine which SQL statements are to be executed, read any CRISP variables necessary to execute the SQL statements, execute the SQL statements, and write back data to the CRISP database. The default value is 50 (centi-seconds) or half a second. If DATAGATE determines that less than half a second has passed from the time it starts processing a logical transaction until the end of that logical transaction is found, then DATAGATE will hibernate for the remaining period.

If users know that their DATAGATE application needs only read the triggers once a minute, then the SET_MIN_TRANSACTION_TIME_TO> opcode can be set to 6000. Less time critical applications can be set accordingly.

A Sample UCF for Rdb/VMS

```
!----------------------------------------------------------------!
!                                                                !
! FILE NAME:              DGT_RDBVMS.UCF                         !
!                                                                !
! LAST MODIFICATION:      20-Jul-1990 10:42                      !
!                                                                !
!                                                                !
! DESCRIPTION:  This is the actual UCF File used to test         !
!               parts of the Rdb/VMS inteface.  It is            !
!               included here for your convenience.              !
!                                                                !
!----------------------------------------------------------------!
!                                                                !
! DEFINITION SECTION:                                            !
!                                                                !
!       The definition section is used to interpret Customer,    !
!       information.  All opcodes documented in this section     !
!       must be present in the UCF file in order for the         !
!       CRISP/PARSER to create the needed global section         !
!       that is used by DATAGATE.                                !
!                                                                !
!----------------------------------------------------------------!

!
DEFINE>       ! Keyword to start definition
              ! section specific processing.

!
!----------------------------------------------------------------!
!                                                                !
! NOTE:                                                          !
!                                                                !
!       The Customer opcodes must be present in the UCF.         !
!       Verify all Customer information for accuracy.           !
!       CRISP/PARSER will display information and errors         !
!       regarding these opcodes.  When customer information      !
!       is incorrect DATAGATE will be run in DEMO mode.          !
!                                                                !
!----------------------------------------------------------------!

!
CUSTOMER_NAME>>                ;
CUSTOMER_LOCATION>>            ;
CUSTOMER_SW_LICENSE>>          ;
CUSTOMER_USE_LIMIT>>           ;
CUSTOMER_CONFIG_CODE>>         ;
CUSTOMER_CONFIG_ID>>           ;
```

A Sample UCF for Rdb/VMS (cont)

```
!
!----------------------------------------------------------------!
!                                                                !
! INITIALIZATION SECTION:                                        !
!                                                                !
!       The initialization section begins with the opcode        !
!       INIT, which is used to define values used by all         !
!       logical transactions.                                    !
!                                                                !
!----------------------------------------------------------------!

!
INIT>
RDBMS_TYPE>                    ; RDBVMS          ! Rdb/VMS
ZERO_DATE>                     ;"13-Jun-1990"    ! For DATE/LONG
!                                                 ! conversions

!
TRANSACTION_TIMEOUT_ACTION>    ; COMMIT          ! on TIMEOUT
TRANSACTION_TIMEOUT>           ; 3600            ! TIMEOUT after
!                                                ! 30 min

!
SET_MIN_TRANSACTION_TIME_TO>   ; 500             ! WAIT 5 secs

!
RDBMS_IDENT>                   ; "DUA0:[RDBUSER.RDBDEMO]PERSONNEL"
STRING_NULL_DATA_VALUE>        ; "Null String"
NUMERIC_NULL_DATA_VALUE>       ; -9999
FLOAT_NULL_DATA_VALUE>         ; -999.99

!
ADB_NODE>                      ; "AZVAX3"        ! CRISP Node
ADB_IDENT>                     ; "TAGTST"        ! CRISP ADB

!
!----------------------------------------------------------------!
!                                                                !
! NOTE:                                                          !
!       The SET_VAR_TO opcodes are used to initialize various    !
!       database variables and triggers.  The PUT_IF and         !
!       GET_IF variables are all of type INTERMEDIATE.  The      !
!       CRISP data types of the other variables can be           !
!       determined by the second character of the variable       !
!       name where L is LONG and S is STRING.                    !
!                                                                !
!----------------------------------------------------------------!
```

A Sample UCF for Rdb/VMS (cont)

```
!
SET_VAR_TO>             RL11000 ; 0
SET_VAR_TO>             RL11001 ; 0
SET_VAR_TO>             RL11010 ; 1
SET_VAR_TO>             WS11010 ; "Starting ...."
SET_VAR_TO>             WS11011 ; "Starting ...."
SET_VAR_TO>             WS11012 ; "Starting ...."
SET_VAR_TO>             WL11000 ; 0
SET_VAR_TO>             WL11010 ; 0
SET_VAR_TO>             WL11011 ; 0
SET_VAR_TO>           GET_IF112 ; 0
SET_VAR_TO>           PUT_IF112 ; 0
SET_VAR_TO>           PUT_IF113 ; 0

!
!----------------------------------------------------------------!
!                                                                !
! PROCESS SECTION:                                               !
!                                                                !
!       The process section begins with the opcode PROCESS.      !
!       It marks the logical end of the definition section       !
!       and begins the process section.  This section is         !
!       used to define all opcodes that DATAGATE is to           !
!       process with every logic scan.                           !
!                                                                !
!----------------------------------------------------------------!

!
PROCESS>

!
!----------------------------------------------------------------!
!                                                                !
! LOGICAL_TRANSACTION:                                           !
!                                                                !
!       The LOGICAL_TRANSACTION opcode is used to mark a         !
!       logical transacation.  One DATAGATE process will         !
!       be started for each LOGICAL_TRANSACTION opcode           !
!       in the file.  A COMMIT or ROLLBACK issued within         !
!       a logical transaction affects all SQL statements         !
!       in the same logical transaction, but does not            !
!       affect SQL statements in other logical                   !
!       transactions.                                            !
!                                                                !
!----------------------------------------------------------------!

!
LOGICAL_TRANSACTION> ;
```

A Sample UCF for Rdb/VMS (cont)

```
!
!----------------------------------------------------------------!
!                                                                !
!       Here is an example of reading several data types         !
!       out of Rdb/VMS into associated CRISP variables.          !
!       Information about a SQL statement starts following       !
!       the LOGICAL_TRANSACTION opcode and ends at the           !
!       EXECUTE type opcode.  A new SQL statement starts         !
!       with the first opcode following an EXECUTE type          !
!       opcode.                                                  !
!                                                                !
!       Notice in the following example that the same DATE       !
!       column is selected twice, once into a CRISP STRING       !
!       variable and next into a CRISP LONG variable.            !
!                                                                !
!----------------------------------------------------------------!

!
ROW_START>                    RL11010 ;
ROW_COUNT>                            ; 1
DISPLAY_NEXT_ROW_INDEX_AT>    RL11010 ;
DISPLAY_END_OF_DATA_AT>     PUT_IF112 ;
DISPLAY_SQL_DONE_AT>        PUT_IF113 ;
DISPLAY_RDBMS_STATUS_AT>      RL11000 ;

!
SQL_STATEMENT>           ; "SELECT LAST_NAME, FIRST_NAME,"

SQL_STATEMENT>           ; "SALARY_START, SALARY_START,"
SQL_STATEMENT>           ; "SALARY_AMOUNT"
SQL_STATEMENT>           ; "INTO #WS11010, #WS11011, #WS11012,"
SQL_STATEMENT>           ; "#WL11011, #WL11010"
SQL_STATEMENT>           ; "FROM CURRENT_SALARY"
SQL_STATEMENT>           ; "WHERE LAST_NAME STARTING WITH 'R'"
SQL_STATEMENT>           ; "ORDER BY LAST_NAME;"

!
EXECUTE_AND_CLR_IF>         GET_IF112 ;

!
!----------------------------------------------------------------!
!                                                                !
!       When the last row is found (DISPLAY_END_OF_DATA_AT>),    !
!       a COMMIT will be issued and this DATAGATE process        !
!       will exit.                                               !
!                                                                !
!----------------------------------------------------------------!

!
COMMIT_AND_CLR_IF>       PUT_IF112 ;
EXIT_IF>                 PUT_IF112 ;
```

A Sample UCF for Rdb/VMS (cont)

```
!
!----------------------------------------------------------------!
!                                                                !
!       This next example takes some of the values read          !
!       from the above table and INSERTs new rows into a         !
!       different table.  Notice too, that the trigger for       !
!       this INSERT statement is set by the                      !
!       DISPLAY_SQL_DONE_AT> opcode for the earlier SELECT       !
!       statement and that when this INSERT statement            !
!       finishes, it sets the trigger so that the above          !
!       SELECT statement retrieves its next row.                 !
!                                                                !
!       Notice also that it does not matter if the               !
!       SQL_STATEMENT> opcodes are before the DISPLAY type       !
!       opcodes or not.  A SQL statement begins immediately      !
!       following the LOGICAL_TRANSACTION opcode until           !
!       an opcode of EXECUTE type is found; then a new SQL       !
!       statement starts.                                        !
!                                                                !
!----------------------------------------------------------------!

!
SQL_STATEMENT>           ; "INSERT INTO TEST_TABLE (NAME_COLUMN,"
SQL_STATEMENT>           ; "DATE_COLUMN, DOUBLE_COLUMN)"
SQL_STATEMENT>           ; "VALUES"
SQL_STATEMENT>           ; "(#WS11010, #WL11011, #WL11010);"

!
ROW_COUNT>                            ; 1
DISPLAY_SQL_DONE_AT>        GET_IF112 ;
DISPLAY_RDBMS_STATUS_AT>      RL11001 ;

!
EXECUTE_AND_CLR_IF>         PUT_IF113 ;
!
!----------------------------------------------------------------!
!                                                                !
!       Display the current scan counter                         !
!                                                                !
!----------------------------------------------------------------!
!
ADD_TO_VAR>                WL11000 ; 1
!
!----------------------------------------------------------------!
!                                                                !
! END SECTION:                                                   !
!                                                                !
!       The end section defines the logical end of the UCF       !
!       file.  Any opcodes or comments declared after this       !
!       opcode are ignored by CRISP/PARSER.                      !
!                                                                !
!----------------------------------------------------------------!
!
END>
```

Notes:

Opcode Definitions

General

The opcodes are defined here in alphabetical order. Unless otherwise noted, the opcode is legal in any of the User Configuration File (UCF) sections defined previously.

> + **Note:**
>
> The parameter values in the following definitions are shown here as decimal values, and are included for use with the Application Database (ADB) variable parameter.

It is not necessary to supply an opcode if the default parameter matches your requirements.

ADB_IDENT>

Opcode which specifies the name or identity of the desired ADB. This opcode MUST be used prior to any ADB variable reference, unless that opcode includes an ADB ID name followed by a colon. The ADB variable portion of the statement is currently not supported. If the default entry is a string then it must be enclosed in double quotes in order to avoid confusion with token definitions or absolute values.

Example: `ADB_IDENT> ;"TRAIN1"  !` Database name

Default: (Not supported)

ADB_NODE>

Opcode which specifies the node where the ADB_IDENT> value is located. The ADB variable portion of the statement is not currently supported. If the ADB_IDENT> opcode is found before an ADB_NODE> opcode is found, then the node for that particular ADB_IDENT> will be the node that DATAGATE is running on.

Example: `ADB_NODE> ; "AZVAX2"  !` Node for TRAIN1 database

Default: Current Node

ADB_TYPE>

Opcode which specifies the Applications Database type. Currently only one type of ADB is supported. The ADB variable portion of the statement is not currently supported.

Example: `ADB_TYPE> ;CRISP32  !` type of database

Default: CRISP32

ADD_TO_VAR>

Opcode which allows the user to add a predetermined value to an ADB memory location. Generally used to increment a pass counter by adding one at the top or bottom of the UCF, but will add any legal value supplied on the right side of the semicolon, including both integer and real values, as well as any of the Tokens defined for any of the other opcodes. Consult the ADB addendums for legal parameter values. The opcode requires a CRISP variable. Regardless of the position of this opcode in the UCF, the add will not take place until the end of a logical transaction pass.

Example: `ADD_TO_VAR> pass_count;  !` Count each pass

Default: 1

BIT_NULL_DATA_VALUE>

Opcode which specifies the value to be written to an INTERMEDIATE variable in the CRISP ADB in the event that a SQL SELECT statement determines that the field in the relational database table associated with the INTERMEDIATE variable has no data. If placed in the INIT> section, the value will be valid for all logical transactions in the UCF, otherwise, there should only be one instance of this opcode per logical transaction. On a logical transaction basis, this is a One Shot opcode.

Example: `BIT_NULL_DATA_VALUE>  ;  1`

Default: 0

BATCH_FILE>

This opcode enables the user to specify the name of a command file which will be submitted to a batch queue (specified by the BATCH_QUE> opcode) when a CRISP variable (specified by the SUBMIT_BATCH_AND_CLR_IF> opcode) is true. The full file specification for the file, including device and directory must be included. In addition, the file specification must be in a quoted string.

Example: `BATCH_FILE> ; "DISK$USER:[RDB.REPORTS]END_OF_SHIFT.COM"`

Default: None

BATCH_QUE>

This opcode enables the user to specify the name of a batch queue to which a command file (specified by the BATCH_FILE> opcode) will be submitted when a CRISP variable (specified by the SUBMIT_BATCH_AND_CLR_IF> opcode) is true. The name of the queue must be in a quoted string.

Example: `BATCH_QUE> ; "REPORTSQUE"`

Default: SYS$BATCH

BATCH_PARAMETER_P1>

This opcode enables is used in conjunction with the SUBMIT_BATCH_AND_CLR_IF> opcode to send a P1 value to the command procedure specified by the BATCH_FILE> opcode. This opcode may have a CRISP variable if desired. If so, the variable must be of CRISP string type.

Example: `BATCH_PARAMETER_P1> ; "Value of P1"`

Default: `" "`

BATCH_PARAMETER_P2>

This opcode enables is used in conjunction with the SUBMIT_BATCH_AND_CLR_IF> opcode to send a P2 value to the command procedure specified by the BATCH_FILE> opcode. This opcode may have a CRISP variable if desired. If so, the variable must be of CRISP string type.

Example: `BATCH_PARAMETER_P2> ; "Value of P2"`

Default: `" "`

BATCH_PARAMETER_P3>

This opcode enables is used in conjunction with the SUBMIT_BATCH_AND_CLR_IF> opcode to send a P3 value to the command procedure specified by the BATCH_FILE> opcode. This opcode may have a CRISP variable if desired. If so, the variable must be of CRISP string type.

Example: `BATCH_PARAMETER_P3> ; "Value of P3"`

Default: `" "`

BATCH_PARAMETER_P4>

This opcode enables is used in conjunction with the SUBMIT_BATCH_AND_CLR_IF> opcode to send a P4 value to the command procedure specified by the BATCH_FILE> opcode. This opcode may have a CRISP variable if desired. If so, the variable must be of CRISP string type.

Example: `BATCH_PARAMETER_P4> ; "Value of P4"`

Default: `" "`

BATCH_PARAMETER_P5>

This opcode enables is used in conjunction with the SUBMIT_BATCH_AND_CLR_IF> opcode to send a P5 value to the command procedure specified by the BATCH_FILE> opcode. This opcode may have a CRISP variable if desired. If so, the variable must be of CRISP string type.

Example: `BATCH_PARAMETER_P5> ; "Value of P5"`

Default: `" "`

BATCH_PARAMETER_P6>

This opcode enables is used in conjunction with the SUBMIT_BATCH_AND_CLR_IF> opcode to send a P6 value to the command procedure specified by the BATCH_FILE> opcode. This opcode may have a CRISP variable if desired. If so, the variable must be of CRISP string type.

Example: `BATCH_PARAMETER_P6> ; "Value of P6"`

Default: `" "`

BATCH_PARAMETER_P7>

This opcode enables is used in conjunction with the SUBMIT_BATCH_AND_CLR_IF> opcode to send a P7 value to the command procedure specified by the BATCH_FILE> opcode. This opcode may have a CRISP variable if desired. If so, the variable must be of CRISP string type.

Example: `BATCH_PARAMETER_P7> ; "Value of P7"`

Default: `" "`

BATCH_PARAMETER_P8>

This opcode enables is used in conjunction with the SUBMIT_BATCH_AND_CLR_IF> opcode to send a P8 value to the command procedure specified by the BATCH_FILE> opcode. This opcode may have a CRISP variable if desired. If so, the variable must be of CRISP string type.

Example: `BATCH_PARAMETER_P8> ; "Value of P8"`

Default: `" "`

CLEAR_STATUS_IF> This opcode must have a CRISP variable. When the value of the variable is non-zero, both the primary and secondary status values are set to zero. This opcode should be used in conjunction with the PRIMARY_STATUS_AT> and the SECONDARY_STATUS_AT> opcodes to aid application logic in handling error conditions.

Example: `CLEAR_STATUS_IF> CLR_STATUS ;`

Default: None

COMMIT_AND_CLR_IF> This opcode must have a CRISP variable. If the value of the variable is non-zero, then a SQL COMMIT will be issued and the value of the CRISP variable will be set to 0. The COMMIT will affect the entire logical transaction.

Example: `COMMIT_AND_CLR_IF> commit_var ;`

Default: None

COMMIT_IF> It is strongly suggested that this opcode have a CRISP variable; if not, a COMMIT will be issued every pass of a logical transaction depending on the default value. If the value of the variable is non-zero, then a COMMIT will be issued. The COMMIT will affect the entire logical transaction.

Example: `COMMIT_IF> commit_var ;`

Default: None

CUSTOMER_CONFIG_CODE>

Opcode which specifies the customer's configuration code to DATAGATE. Must be entered exactly as specified on your licensing agreement. Must be specified before the process section, immediately following the CUSTOMER_USE_LIMIT> opcode and immediately preceding the CUSTOMER_CONFIG_ID> opcode in the DEFINE> section of the UCF.

CUSTOMER_CONFIG_ID>

Opcode which specifies the customer's configuration identifier to DATAGATE. Must be entered exactly as specified on your licensing agreement. Must be specified before the process section, immediately following the CUSTOMER_CONFIG_CODE> opcode in the DEFINE> section of the UCF.

CUSTOMER_LOCATION>

Opcode which specifies the customer's configuration location to DATAGATE. Must be entered exactly as specified on your licensing agreement, enclosed in double quotes (" "). Must be specified before the process section, immediately following the CUSTOMER_NAME> opcode and immediately preceding the CUSTOMER_SW_LICENSE> opcode in the DEFINE> section of the UCF.

CUSTOMER_NAME>

Opcode which specifies the customer's configuration name to DATAGATE. Must be entered exactly as specified on your licensing agreement, enclosed in double quotes (" "). Must be specified before the process section, immediately preceding the CUSTOMER_LOCATION> opcode and immediately following the DEFINE> opcode in the UCF.

CUSTOMER_SW_LICENSE>

Opcode which specifies the customer's configuration software license code to DATAGATE. Must be entered exactly as specified on your licensing agreement. Must be specified before the process section, immediately following the CUSTOMER_LOCATION> opcode and immediately preceding the CUSTOMER_USE_LIMIT> opcode in the DEFINE> section of the UCF.

CUSTOMER_USE_LIMIT>

Opcode which specifies the customer's configuration use limit code to DATAGATE. Must be entered exactly as specified on your licensing agreement. Must be specified before the process section, immediately following the CUSTOMER_SW_LICENSE> opcode and immediately preceding the CUSTOMER_CONFIG_CODE> opcode in the DEFINE> section of the UCF.

DEFINE>

Opcode which designates the start of the definition section. All opcodes in this section are automatically 'one shot' regardless of the number of right angle brackets supplied.

Example: `DEFINE>` ! Begin definition section

Default: (No parameters)

DISABLE_RDBMS_COMMAND>

Opcode specifying which of the RDBMS commands are to be disabled and prevented from executing. If an RDBMS command has been disabled, then any SQL statement found which begins with the command will not be executed. Instead, an error will be generated. This opcode may be placed in the Init Section or the Process Section.

Parameter	Token
100	ALTER
101	COMMENT
102	CREATE
103	DELETE
104	DROP
105	GRANT
106	INSERT
107	REVOKE
139	SELECT
109	UPDATE

Example: `DISABLE_RDBMS_COMMAND> ; SELECT !` Disable SELECTs

Default: `ALTER, COMMENT, DELETE, DROP, GRANT, REVOKE and UPDATE`

DISPLAY_END_OF_DATA_AT>

This opcode only has meaning for SQL SELECT statements; it requires a CRISP variable; any default value will be ignored. When the last row in a query has been returned, the DATAGATE will write a value of 1 to the variable, otherwise a value of 0 will be written. This value is only written if there were no errors in the SQL statement that was executed. There may be one DISPLAY_END_OF_DATA_AT> opcode per execute type opcode (EXECUTE_IF>, EXECUTE_COMMIT_IF>, EXECUTE_AND_CLR_IF>, or EXECUTE_COMMIT_AND_CLR_IF>).

Example: `DISPLAY_END_OF_DATA_AT> EOD  ;  !` End of data?

Default: Not applicable

DISPLAY_MORE_DATA_AT>

This opcode only has meaning for SQL SELECT statements; it requires a CRISP variable; any default value will be ignored. If the last row in a query has not been returned, the DATAGATE will write a value of 1 to the variable, otherwise, when the last row has been returned, a value of 0 will be written. This value is only written if there were no errors in the SQL statement that was executed. There may be one DISPLAY_MORE_DATA_AT> opcode per execute type opcode (EXECUTE_IF>, EXECUTE_COMMIT_IF>, EXECUTE_AND_CLR_IF>, or EXECUTE_COMMIT_AND_CLR_IF>).

Example: `DISPLAY_MORE_DATA_AT> EOD` ; ! More rows?

Default: Not applicable

DISPLAY_NEXT_ROW_INDEX_AT>

This opcode only has meaning for SQL SELECT statements; it requires a CRISP variable; any default value will be ignored. It is used in conjunction with the ROW_START> opcode to easily retrieve rows in a table by groups. For example, if the row start value is 1, and the row count value is 5, DATAGATE will return rows 1 through 5 as found in the query. The next row index value of 6 will be written to the variable associated with this opcode. When all rows in a query have been retrieved, the default value found for the ROW_START> opcode will be written to the variable associated with this opcode. It is suggested that the variable for the DISPLAY_NEXT_ROW_INDEX_AT> opcode and the variable for the ROW_START> opcode be the same. This value is only written if there were no errors in the SQL statement that was executed. There may be one DISPLAY_NEXT_ROW_INDEX_AT> opcode per execute type opcode (EXECUTE_IF>, EXECUTE_COMMIT_IF>, EXECUTE_AND_CLR_IF>, or EXECUTE_COMMIT_AND_CLR_IF>).

Example: `DISPLAY_NEXT_ROW_INDEX_AT>ROW_START_VAR ;`

Default: Not applicable

DISPLAY_RDBMS_STATUS_AT>

This opcode provides users with a means to determine the RDBMS specific status returned by the RDBMS when a SQL statement is executed; it requires a CRISP variable. It is only returned when the statement is executed, that is with the EXECUTE_IF>, EXECUTE_CLR_IF>, EXECUTE_COMMIT_IF>, and the EXECUTE_COMMIT_AND_CLR_IF> opcodes. (The status will not be written for the COMMIT_IF> and ROLLBACK_IF> opcodes). For Rdb/VMS, the value returned will be the value of RDB$LU_STATUS which is returned in the RDB$MESSAGE_VECTOR by SQL. The values of RDB$LU_STATUS follow the usual VAX/VMS standards for status values. (See the Rdb/VMS Guide to Using SQL for detailed information.) This value is written regardless of whether any errors were found during execution of the SQL statement. There may be one DISPLAY_RDBMS_STATUS_AT> opcode per execute type opcode (EXECUTE_IF>, EXECUTE_COMMIT_IF>, EXECUTE_AND_CLR_IF>, or EXECUTE_COMMIT_AND_CLR_IF>).

Example: `DISPLAY_RDBMS_STATUS_AT> RDBMS_STATUS_VAR ;`

Default: Not applicable

DISPLAY_SQLCODE_AT>

This opcode provides users with a means to determine the SQL status returned by the RDBMS when a SQL statement is executed; it requires a CRISP variable. It is only returned when the statement is executed, that is with the EXECUTE_IF>, EXECUTE_CLR_IF>, EXECUTE_COMMIT_IF>, and the EXECUTE_COMMIT_AND_CLR_IF> opcodes. (The status will not be written for the COMMIT_IF> and ROLLBACK_IF> opcodes). The value returned will be the SQLCODE value for the particular RDBMS. While exact values are RDBMS specific, in general, a value of 0 means success, a positive value is a warning or exception condition, and a negative value is an error. This value is written regardless of whether any errors were found during execution of the SQL statement. There may be one DISPLAY_SQLCODE_AT> opcode per execute type opcode (EXECUTE_IF>, EXECUTE_COMMIT_IF>, EXECUTE_AND_CLR_IF>, or EXECUTE_COMMIT_AND_CLR_IF>).

Example: `DISPLAY_SQLCODE_AT> SQLCODE_VAR ;`

Default: Not applicable

DISPLAY_SQL_DONE_AT>

This opcode must have a CRISP variable. DATAGATE will write a value of 1 to the variable when a SQL statement is finished executing. Users may use this feature to "chain" SQL statements, that is to trigger SQL statement 2 when SQL statement 1 is finished. This value is only written if there were no errors in the SQL statement that was executed. There may be one DISPLAY_SQL_DONE_AT> opcode per execute type opcode (EXECUTE_IF>, EXECUTE_COMMIT_IF>, EXECUTE_AND_CLR_IF>, or EXECUTE_COMMIT_AND_CLR_IF>).

Example: `DISPLAY_SQL_DONE_AT> NEXT_TRIGGER_VAR ;`

Default: None

DISPLAY_TOTAL_ROWS_AT>

This opcode only has meaning for SQL SELECT statements; it requires a CRISP variable; any default value will be ignored. DATAGATE will write the total number of rows written for a single query. For instance, the user requests all rows in a table, 5 at a time. After the first execution of the query, a value of 5 is written to the DISPLAY_TOTAL_ROWS_AT> variable; after the second execution, a value of 10 is written. This continues until the last row is found. This value is only written if there were no errors in the SQL statement that was executed. There may be one DISPLAY_TOTAL_ROWS_AT> opcode per execute type opcode (EXECUTE_IF>, EXECUTE_COMMIT_IF>, EXECUTE_AND_CLR_IF>, or EXECUTE_COMMIT_AND_CLR_IF>).

Example: `DISPLAY_TOTAL_ROWS_AT> TOTAL_ROWS_VAR ;`

Default: Not applicable

ENABLE_RDBMS_COMMAND>

Opcode which specifies which of the RDBMS commands are to be enabled and allowed to execute. Unless an RDBMS command has been enabled, any SQL statement found which begins with the command will not be executed. Instead, an error will be generated.

Parameter	Token
100	ALTER
101	COMMENT
102	CREATE
103	DELETE
104	DROP
105	GRANT
106	INSERT
107	REVOKE
139	SELECT
109	UPDATE

Example: `ENABLE_RDBMS_COMMAND> ; CREATE !` Enable CREATEs

Default: INSERT and SELECT

END>

Opcode which signifies the end of the UCF file, and thus the end of the logical process path. Anything following this opcode is ignored.

Example: `END>  !` End of processing

Default: (No parameters)

EXECUTE_AND_CLR_IF>

This opcode requires a CRISP variable; any default value found will be ignored. The variable associated with this opcode is called a "trigger" variable. When its value is non-zero, the associated SQL statement will be executed. If no errors are found during execution, DATAGATE will then change the value of the variable to zero.

Example: `EXECUTE_AND_CLR_IF> EXECUTE_VAR ;`

Default: None

EXECUTE_COMMIT_AND_CLR_IF>

This opcode requires a CRISP variable; any default value found will be ignored. The variable associated with this opcode is called a 'trigger' variable. When its value is non-zero, the associated SQL statement will be executed. If no errors are found during the execution, a SQL COMMIT will be issued and DATAGATE will then change the value of the variable to zero.

Example: `EXECUTE_COMMIT_AND_CLR_IF> EXECUTE_VAR` ;

Default: None

EXECUTE_COMMIT_IF>

It is strongly suggested that this opcode have a CRISP variable. If so, the variable associated with this opcode is called a 'trigger' variable. When its value is non-zero, the associated SQL statement will be executed. If no errors are found during the execution, a SQL COMMIT will be issued.

Example: `EXECUTE_COMMIT_IF> EXECUTE_VAR` ;

Default: None

EXECUTE_IF>

It is strongly suggested that this opcode have a CRISP variable. If so, the variable associated with this opcode is called a "trigger" variable. When its value is non-zero, the associated SQL statement will be executed.

Example: `EXECUTE_IF> EXECUTE_VAR` ;

Default: None

EXIT_IF>

Opcode for user to supply an ADB variable or a default value, which will allow a conditional exit, on a Logical Transaction basis. A manual restart will be required after such an exit. Legal values are zero (0) and non-zero. Any supplied default value is only used if no ADB variable name is supplied or if, for some reason, the system is unable to access the ADB variable. Since a non-zero value will force the DATAGATE task to exit, not supplying an ADB variable is usually not meaningful.

Example: `EXIT_IF> EXIT_1; !` DATAGATE to exit?

Default: 0

FLOAT_NULL_DATA_VALUE>

Opcode which specifies the value to be written to a FLOAT variable in the CRISP ADB in the event that a SQL SELECT statement determines that the field in the relational database table associated with the FLOAT variable has no data. If placed in the INIT> section, the value will be valid for all logical transactions in the UCF, otherwise, there should only be one instance of this opcode per logical transaction. On a logical transaction basis, this is a One Shot opcode.

Example: `FLOAT_NULL_DATA_VALUE> FLOAT_NULL_VAR ;`

Default: -999.0

INIT>

Optional opcode which designates the start of the initialization section. All opcodes in this section are automatically "one shot" regardless of the number of right angle brackets supplied.

Example: `INIT> !` Begin initialization section

Default: (No parameters)

LOGICAL_TRANSACTION>

Opcode which specifies the start of a logical transaction. One DATAGATE process will be started for each LOGICAL_TRANSACTION> opcode located in the UCF. At least one LOGICAL_TRANSACTION> opcode must be located.

Example: `LOGICAL_TRANSACTION> !` Begin a logical transacation

Default: (No parameters)

NULLS_CHECKING>

Opcode used if NULL values are to be written to the relational database. Tokens of ON and OFF are provided.

When NULLS checking is ON, DATAGATE compares the value found in Crisp to the null data value for that data type. If the values match, then a NULL is written to the relational database. (Null data values are defined using the STRING_NULL_DAT_VALUE>, NUMERIC_NULL_DATA_VALUE>, FLOAT_NULL_DATA_VALUE>, and BIT_NULL_DATA_VALUE> opcodes.)

If NULLS checking is OFF, then no comparisons are made and the value found in Crisp is what is written to the relational database.

Example: `NULLS_CHECKING> ; ON`

Default: OFF.

NUMERIC_NULL_DATA_VALUE>

Opcode which specifies the value to be written to a NUMERIC or LONG variable in the CRISP ADB in the event that a SQL SELECT statement determines that the field in the relational database table associated with the NUMERIC or LONG variable has no data. If placed in the INIT> section, the value will be valid for all logical transactions in the UCF, otherwise, there should only be one instance of this opcode per logical transaction. On a logical transaction basis, this is a One Shot opcode.

Example: `NUMERIC_NULL_DATA_VALUE> ; -99`

Default: -999.0

PRIMARY_STATUS_AT>

There may be exactly one instance of this opcode per Logical Transaction. At the end of each pass, DATAGATE writes the current process status to the CRISP variable that is named with this opcode. While this opcode is not required, it is strongly recommended that this opcode be used. Application logic may then be written to handle error conditions. A value of 0 signifies that no error condition has been encountered. PRIMARY_STATUS_AT> may be used in conjunction with SECONDARY_STATUS_AT> and CLEAR_STATUS_IF> to determine reasons for errors. For more detailed information, refer to the appropriate sections of this manual. The following table shows the values that DATAGATE will write to the primary status variable.

Status Value	Meaning of error
0	No Error Occurred
100	Batch Job Error
125	RDBMS_COMMAND> Error
150	A CRISP _ _ - _> RDBMS Data Type Conversion Error
175	Invalid Date String Located in ZERO_DATE>
200	DATAGATE Exited
225	Version Mismatch Error; DATAGATE Exited
250	Relational Database Not Located
275	A SQL Statement Was Not Executed
300	RDBMS_TYPE> Not Located
325	SQL Statement Parse Error
350	Error in Reading From CRISP
375	RDBMS SQL Error
400	Error in Clearing a Trigger
425	VMS Error
450	WORF Error
475	Error in Writing to CRISP

Example: `PRIMARY_STATUS_AT> DATAGATE_STATUS ;`

Default: None.

PROCESS>

Opcode which signifies the start of the process section. Unless otherwise specified, by the use of two right angle brackets, all opcodes following this opcode are done each pass.

Example: `PROCESS> !` Begin process section

Default: (No parameters)

RDBMS_IDENT>

For Rdb/VMS this opcode specifies the name of the relational database that DATAGATE is to use. This opcode may NOT specify a CRISP variable. The value of this opcode must be a full file specification which can contain only those logicals in the system table. The value must be enclosed in double quotes. The file extension should be omitted. There is no default value; this opcode is required.

Example: `RDBMS_IDENT> ; "DUA0:[DATABASE]PERSONNEL"`

Default: None

RDBMS_TYPE>

This opcode specifies the Relational Database Management System (RDBMS)product (ie, Rdb/VMS, Oracle, Ingres) that the user has on his system. If used, this opcode may not specify a CRISP variable. Currently, Rdb/VMS is the only RDBMS implemented.

Parameter	Token
100	RDBVMS
200	ORACLE
300	INGRES

Example: `RDBMS_TYPE> ; ORACLE`

Default: `RDBVMS`

ROLLBACK_AND_CLR_IF>

This opcode requires a CRISP variable; any default value will be ignored. When the value of the variable is non-zero, a SQL ROLLBACK statement will be issued. If no errors are found, the variable will be set to zero. In general, it is expected that this opcode will be useful in handling error conditions.

Example: `ROLLBACK_AND_CLR_IF> ERROR_VAR ;`

Default: None

ROLLBACK_IF>

This opcode requires a CRISP variable; any default value will be ignored. When the value of the variable is non-zero, a SQL ROLLBACK statement will be issued. In general, it is expected that this opcode will be useful in handling error conditions.

Example: `ROLLBACK_IF> ERROR_VAR ;`

Default: None

ROW_COUNT>

This opcode is useful when performing SQL INSERT and SELECT statements. Its value (either from a variable, or using the default) is used in conjunction with CRISP arrays. Rather then writing the same INSERT statement five times in the UCF, where the only difference from one INSERT statement to the next is the array index (i.e., ARRAY(0), ARRAY(1), ARRAY(2), ARRAY(3), ARRAY(4)), the user may write the first element of the array in the INSERT statement (ARRAY[0]), and then specify a row count value of 5. DATAGATE will then take 5 values starting with the element specified in the SQL statement and create 5 rows to store in the RDBMS database. With SELECT statements, the user can again retrieve more than one row at a time, by using the ROW_COUNT> opcode to tell DATAGATE how many array elements are available to hold rows returned from the query. Any value for this opcode will be ignored unless its associated SQL statement is either an INSERT or a SELECT.

Example: `ROW_COUNT> ROW_COUNT_VAR ;`

Default: 1

ROW_START>

This opcode is used in conjunction with the ROW_COUNT> opcode when retrieving multiple rows in the same SQL SELECT query. For instance, suppose there are 100 rows in a table, but you only have room for 5 of them in the CRISP database. Specify Row Count as 5 (how much room is in the database) and Row Start as 1 (meaning, return to me 5 rows starting with row 1). This returns rows 1 to 5 in the table. Now, change the Row Start to 6 and set the trigger associated with the SQL statement. DATAGATE will now return 5 rows starting at row 6 (i.e., rows 6 to 10). Use this opcode in conjunction with the DISPLAY_NEXT_ROW_INDEX_AT> opcode to easily "scroll" through all rows in a table. (Make sure however, that DATAGATE does not issue a COMMIT or ROLLBACK until all data has been returned.)

Example: `ROW_START> ROW_START_VAR ;`

Default: 1

SECONDARY_STATUS_AT>

This opcode is used in conjunction with PRIMARY_STATUS_AT> and CLEAR_STATUS_IF> to aid application logic in determining DATAGATE errors. A value of 0 means that no error has occurred. The value that DATAGATE writes to the Secondary Status variable is dependant on the value of the Primary Status. Refer to the appropriate sections of this manual for more information.

Example: `SECONDARY_STATUS_AT> SECONDARY_STATUS ;`

Default: None.

SET_MIN_TRANSACTION_TIME_TO>

Opcode which sets the Minimum Transaction Time on a Logical Transaction basis. The Mimimun Transaction Time is the minimum amount of time that a single DATAGATE process must take to perform its own Logical Transaction. If, after all SQL statement triggers are read, and any SQL statements executed, the elapsed time is less than this value, the DATAGATE process will delay for the remaining time. The Minimum Transaction Time is always specified in centi-second units (1/100 of a second). A value of 0 will disable this feature. This feature will aid users in monitoring the CPU usage on the RDBMS VAX. The supplied default value is 50 (or 1/2 a second).

Example: `SET_MIN_TRANSACTION_TIME_TO> ; 3600 !` One minute

Default: 50 (1/2 second)

SET_VAR_TO>

Opcode which allows the user to place a predetermined value into an ADB variable location. Generally used to clear status or edge trigger controls. Will accept both integer and real values, as well as any of the Tokens defined for any of the other opcodes. Certain ADB supports may require specific memory types or locations. Consult the ADB addendums for legal parameter values. Also consult the Token Definition Addendum, and any Opcode Definitions allowing for tokens.

Example: `SET_VAR_TO> P_COUNT;73 !` PASS_COUNT set to 73

Default: 0

SQL_STATEMENT>

Opcode used to actually state what SQL statement is to be executed. The SQL statement may be found in the default string, or in a CRISP variable. When possible, put the SQL statement in the default string, because taking SQL statements from CRISP variables is slower and has a higher overhead. Only put SQL statements in CRISP variables when the query must be "adhoc". For known data transfers, use the default string. Because SQL statements in general are too long to fit on a single line in the UCF, multiple consecutive SQL_STATEMENT> opcodes are allowed. DATAGATE will continue parsing the values found for the SQL_STATEMENT opcodes until a semi-colon (;) terminator is found as the last character in either a CRISP variable or the default string. This semi-colon should not be confused with the semi-colon separating the CRISP variable from the default value in the opcode syntax. Note in the example that the # precedes the names of the CRISP variables whose values are to be placed in the SQL statement at execution time.

Example:
```
SQL_STATEMENT> ; "INSERT INTO EMPLOYEES"
SQL_STATEMENT> ; "(LAST_NAME,FIRST_NAME)"
SQL_STATEMENT> ; "VALUES"
SQL_STATEMENT> ; "(#LAST_VAR, #FIRST_VAR);"
```

Default: None

STRING_NULL_DATA_VALUE>

Opcode which specifies the value to be written to a STRING variable in the CRISP ADB in the event that a SQL SELECT statement determines that the field in the relational database table associated with the STRING variable has no data. If placed in the INIT> section, the value will be valid for all logical transactions in the UCF, otherwise, there should only be one instance of this opcode per logical transaction. On a logical transaction basis, this is a One Shot opcode.

Example: `STRING_NULL_DATA_VALUE> ; "Null Data Found"`

Default: "" (A zero length string)

SUBMIT_BATCH_AND_CLR_IF>

This opcode requires a CRISP variable; any default value found will be ignored. When the value of this variable is non-zero, the command procedure (specified by the BATCH_FILE> opcode) will be submitted to a batch queue (specified by the BATCH_QUE> opcode) with a log file with the same name as the batch file with an extension of `.LOG`.

Example: `SUBMIT_BATCH_AND_CLR_IF> END_OF_SHIFT_VARIABLE ;`

Default: None

TIMEOUT>

Opcode which specifies how long DATAGATE will wait until a default COMMIT or ROLLBACK will be issued. Normally, users will use the EXECUTE_COMMIT_IF> or EXECUTE_COMMIT_AND_CLR_IF> opcodes to execute a SQL statement and then issue a commit. However, users may also use the EXECUTE_IF> and EXECUTE_AND_CLR_IF> opcodes in which a case a COMMIT is not automatically issued. Also, if the execution of a SQL statement fails, a COMMIT will not be issued even if the EXECUTE_COMMIT_IF> or EXECUTE_COMMIT_AND_CLR_IF> opcodes were used. Once a SQL statement is executed, the RDBMS puts locks on tables and rows according to the type of SQL statement. It is not good relational database practice to leave the database in this condition for long periods of time. This opcode gives the user the flexibility to state exactly how long from the start of a SQL statement DATAGATE will wait before issuing either a COMMIT or a ROLLBACK. The TIMEOUT value is in units of seconds. The default value is 3600 seconds (30 minutes).

Example: `TIMEOUT> ; 300 !` Timeout after 5 minutes

Default: 3600 (30 minutes)

TIMEOUT_ACTION>

This opcode gives the user the ability to specify which command (COMMIT or ROLLBACK) DATAGATE will issue if the timeout described in TIMEOUT> occurs. The default action is ROLLBACK.

Parameter	Token
100	COMMIT
200	ROLLBACK

Example: `TIMEOUT_ACTION> ; COMMIT`

Default: `ROLLBACK`

ZERO_DATE>

This opcode is only useful if users will be retrieving RDBMS DATE values and writing them to CRISP LONG variables or vice versa. An RDBMS DATE to CRISP LONG conversion involves taking the DATE value and converting it to the number of seconds since the zero date. A default date of "01-APR-1970 00: 00: 00.00" has been provided. If a CRISP variable is used in the ZERO_DATE> opcode, it must be of type STRING. Also, the data contained in the string must be in the VAX date format of

"DD-MON-YYYY HH:MM:SS.SS"

If any portion of the VAX date string is missing, then the values from the default string will be used. For example, if the user wishes the zero date to be "14-DEC-1970 00:00:00.00", the string need only contain the value "14-DEC"

Example: `ZERO_DATE> ; "01-JAN-1990 12:00:00.00"`

Default: `"01-APR-1970 00:00:00.00"`

Token Definitions

General

The keywords (called tokens), which may be used in place of literal numeric values, are listed here in first alphabetical, and then numerical order. Refer to the specific opcode for more details. This listing is meant merely as a summary of the tokens that are legal with the various opcodes.

+ **NOTE:**

The ADD_TO_VAR> and SET_VAR_TO> opcodes may use any token.

Token Definitions - Alphabetical List

Token	Value	Opcode(s)
ALTER	100	ENABLE_RDBMS_COMMAND> DISABLE_RDBMS_COMMAND>
COMMENT	101	ENABLE_RDBMS_COMMAND> DISABLE_RDBMS_COMMAND>
COMMIT	100	TIMEOUT_ACTION>
CRISP32	0	ADB_TYPE>
CREATE	102	ENABLE_RDBMS_COMMAND> DISABLE_RDBMS_COMMAND>
DELETE	103	ENABLE_RDBMS_COMMAND> DISABLE_RDBMS_COMMAND>
DROP	104	ENABLE_RDBMS_COMMAND> DISABLE_RDBMS_COMMAND>
GRANT	105	ENABLE_RDBMS_COMMAND> DISABLE_RDBMS_COMMAND>
INGRES	300	RDBMS_TYPE>
INSERT	106	ENABLE_RDBMS_COMMAND> DISABLE_RDBMS_COMMAND>
RDBVMS	100	RDBMS_TYPE>
REVOKE	107	ENABLE_RDBMS_COMMAND> DISABLE_RDBMS_COMMAND>
ROLLBACK	200	TIMEOUT_ACTION>
SELECT	139	ENABLE_RDBMS_COMMAND> DISABLE_RDBMS_COMMAND>
UPDATE	109	ENABLE_RDBMS_COMMAND> DISABLE_RDBMS_COMMAND>

Token Definitions - Numerical List

Token	Value	Opcode(s)
CRISP32	0	ADB_TYPE>
ALTER	100	ENABLE_RDBMS_COMMAND> DISABLE_RDBMS_COMMAND>
COMMIT	100	TIMEOUT_ACTION>
RDBVMS	100	RDBMS_TYPE>
COMMENT	101	ENABLE_RDBMS_COMMAND> DISABLE_RDBMS_COMMAND>
CREATE	102	ENABLE_RDBMS_COMMAND> DISABLE_RDBMS_COMMAND>
DELETE	103	ENABLE_RDBMS_COMMAND> DISABLE_RDBMS_COMMAND>
DROP	104	ENABLE_RDBMS_COMMAND> DISABLE_RDBMS_COMMAND>
GRANT	105	ENABLE_RDBMS_COMMAND> DISABLE_RDBMS_COMMAND>
INSERT	106	ENABLE_RDBMS_COMMAND> DISABLE_RDBMS_COMMAND>
REVOKE	107	ENABLE_RDBMS_COMMAND> DISABLE_RDBMS_COMMAND>
UPDATE	109	ENABLE_RDBMS_COMMAND> DISABLE_RDBMS_COMMAND>
SELECT	139	ENABLE_RDBMS_COMMAND> DISABLE_RDBMS_COMMAND>
ROLLBACK	200	TIMEOUT_ACTION>
INGRES	300	RDBMS_TYPE>

Notes:

Status and Error Reporting

General

Each DATAGATE process maintains two process-wide values: Primary Status and Secondary Status. These two values are initialized to 0 when the DATAGATE process starts, and remain 0 unless an erro occurs. When an error occurs, both values are updated according to the type of error condition detected and are not updated again until another error condition occurs, or the user specifically clears them using the CLEAR_STATUS_IF> opcode.

DATAGATE writes these values to the specified application database if the PRIMARY_STATUS_AT> and/or SECONDARY_STATUS_AT> opcodes are included in the UCF. Based on the values that DATAGATE returns to the CRISP variables, application logic may respond to DATAGATE errors. Once an error has been handled by the CRISP application, the DATAGATE status values may be cleared.

In addition to the process-wide status values, DATAGATE will report SQL erros on a SQL statement basis. Each time a SQL statement is executed, DATAGATE writes requested values back to a CRISP database. The user may request the SQLCODE value (via the DISPLAY_SQLCODE_AT> opcode) and/or the RDBMS specific status (via the DISPLAY_RDBMS_STATUS_AT> opcode). These values will change only when a trigger goes from false to true.

Regardless of whether status values are written to the CRISP application database, DATAGATE signals both informational and error messages. For DATAGATE batch processes, these messages are located in the DATAGATE log file. If DATAGATE is running on a CRISP VAX, these messages are also signalled to the CRISP console.

Primary Status Code Definitions

The following is a list of implemented Primary Status codes and their
definitions.

Code	Definition
100	BATCH_ERROR - An error was found when attempting to submit a batch job. The Secondary Status value will be FILE_NOT_FOUND or FILE_TYPE.
112	BOGUS_TRIGGER - An EXECUTE type opcode was found with no corresponding SQL_STATEMENT opcode. The Secondary Status value is an index into the UCF.
125	COMMAND_ERROR - An invalid value was found for the ENABLE_RDBMS_COMMAND> or the DISABLE_RDBMS_COMMAND> opcodes. The Secondary Status value is an index into the UCF.

(Continued on next page.)

Primary Status Code Definitions (cont)

Code	Definition
137	COMMIT_ERROR - An error occured while issuing a COMMIT. In this case, the Secondary Status is the SQLCODE value.
150	CONVERSION_ERROR - A data type conversion error was detected when attempting to convert from the RDBMS to CRISP or vice versa. The Secondary Status value is an index into the UCF.
175	DATE_ERROR - An invalid date string was detected for the ZERO_DATE> opcode or an error occurred when an attempt was made to read/write a DATE column in the relational database. The Secondary Status value is an index into the UCF.
200	EXIT - This value will be written when the DATAGATE process exits, but only if the Primay Status value is 0. The Secondary Status value will also be EXIT, but only if the Secondary Status value is 0.
225	MISMATCH - The number of CRISP variables did not match the number of relational database columns in a SQL statement. The Secondary Status value is an index into the UCF.
250	NO_DATABASE - DATAGATE was unable to locate the specified relational database. In this case, the Secondary Status is the SQLCODE value.
275	NO_EXECUTE - Due to errors, a SQL statement was not executed. The Secondary Status value is an index into the UCF.
300	NO_INIT - DATAGATE attempted RDBMS calls before the RDBMS_TYPE> opcode was located. Make sure that the RDBMS_TYPE> opcode is located in the UCF before the RDBMS_INDENT> opcode and any SQL_STATEMENT> opcodes. The Secondary Status value is an index into the UCF.
325	PARSE_ERROR - An error was detected when parsing a SQL statement. The Secondary Status value is an index into the UCF.
350	READ_ERROR - An error was detected when reading CRISP database trigger or when reading data to be placed in a SQL statement. Secondary Status values will be NO_DB, NO_NODE, NO_VARIABLE, or OTHER.
367	ROLLBACK_ERROR - An error occurred while issuing a ROLLBACK. In this case, the Secondary Status is the SQLCODE value.

(Continued on next page.)

Primary Status Code Definitions (cont)

375	SQL_ERROR - The RDBMS returned an error following execution of a SQL statement. The Secondary Status value is an index into the UCF.
400	SYMBOL_WRITE_ERROR - An error was detected when clearing a trigger. The Secondary Status value is an index into the UCF.
412	TRANSACTION_ERROR - Internal error. Contact CRISP Automation Systems. The Secondary Status will contain a value to be reported.
425	VMS_ERROR - An unexpected VMS error was detected. In this case, the Secondary Status value will be the VMS status value unless the problem is Insufficent Memory in which case the value will be INSFMEM.
450	WORF_ERROR - An unexpected WORF error was detected. The Secondary Status value will be the WORF status value.
475	WRITE_ERROR - An error was detected when performing CRISP database writes. Secondary Status values will be NO_DB, NO_NODE, NO_VARIABLE, or OTHER.

Secondary Status Code Definitions

The following is a list of implemented Secondary Status codes and their
definitions.

Code	**Definition**
1000	FILE_NOT_FOUND - The file specified in the BATCH_FILE> opcode was not located.
1025	FILE_TYPE - The file specified in the BATCH_FILE> opcode was not of type `.COM`.
1050	INSFMEM - DATAGATE has insufficient memeory. DATAGATE will exit.
1075	NO_DB - CRISP database not located.
1100	NO_NODE - CRISP node not located.
1125	NO_VARIABLE - CRISP variable not located.
1150	OTHER - Something besides NO_DB, NO_VARIABLE, and NO_NODE.

RDBMS Status Code Definitions

Status codes in the DATAGATE product are those of the users Relational Database Management System of choice. These codes will be returned to the ADB through the DISPLAY_SQLCODE_AT> and DISPLAY_RDBMS_STATUS_AT> opcodes.

The DISPLAY_SQLCODE_AT> opcode cause the RDBMS value of SQLCODE to be returned. While a SQLCODE value itself is RDBMS specific, in general, a SQLCODE value of 0 means success; a positive value is a warning; a negative value is an error. For more detailed information, refer to the values in the RDBMS documentation.

For Rdb/VMS the DISPLAY_RDBMS_STRING_AT> opcode causes the value of RDB$MESSAGE_VECTOR to be written to the specified variable. For more detailed information refer to the Rdb/VMS Guide to Using SQL.

Glossary

General

Following are explanations to technical terms, acronyms, and mnemonics used throughout this document.

ADB	Application Database -- a database, for example the CRISP database
Opcode	Operation Code, the function to perform
RDBMS	Relational Database Management System. A generic term for a relational database product.
SQL	Structured Query Language
Token	Keyword, predefined value
UCF	User Configuration (or Control) File

Notes: