

---

# **Warehouse Reference Manual**

**Taurus Software, Inc.**

Taurus Software Inc., 420 Brewster Avenue, Redwood City, CA, 94063  
(650) 482-2022  
(650) 482-2010 FAX  
support@taurus.com

---

First Edition, version 2.....October 1994  
Edition 2, version 2.....February 1995  
Edition 3, version 2.....April 1995  
Edition 4, version 2.....July 1996  
Edition 5, version 2.....October 1996  
Edition 6, version 2.....March 1997  
Edition 7, version 2.....July 1998  
Edition 8, version 2.....March 1999  
Edition 9, version 2.07.....November 1999  
Edition 10, version 2.07.....April 2003  
Edition 11, version 3.02.....June 2008

## **NOTICE**

The information contained in this document is subject to change without notice.

Taurus Software, Inc. makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Taurus Software, Inc. shall not be liable for errors herein or for incidental or consequential damages in connection with the use of this material.

This document contains proprietary information which is protected by copyright. All rights are reserved. No part of this document may be reproduced or translated in any form without the prior written consent of Taurus Software, Inc.

© 1989-2008 by Taurus Software, Inc.

---

---

# Table of Contents

<b>CHAPTER ONE: INTRODUCTION</b>	<b>1</b>
PRODUCT OVERVIEW	2
HOW TO USE THIS MANUAL	2
PRODUCT VERSION	5
<b>CHAPTER TWO: WAREHOUSE SCRIPTS</b>	<b>7</b>
SCRIPTING LANGUAGE OVERVIEW	8
<b>CHAPTER THREE: WAREHOUSE STATEMENTS</b>	<b>13</b>
CALL STATEMENT	16
CLOSE STATEMENT	18
COMMIT STATEMENT	20
COPY STATEMENT	23
CREATE STATEMENT	30
DEFINE STATEMENT	32
DELETE STATEMENT	38
DIRECT STATEMENT	39
END STATEMENT	41
ESCAPE STATEMENT	42
EXIT STATEMENT	43
FORMAT STATEMENT	44
FUNCTION STATEMENT	47
GO STATEMENT	53
HEADER STATEMENT	54
HELP STATEMENT	57
IF STATEMENT	58
LIST STATEMENT	60
OPEN STATEMENT	63
PRINT STATEMENT	66
READ STATEMENT	71
RETURN STATEMENT	78
ROLLBACK STATEMENT	80
SET STATEMENT	82
SETVAR STATEMENT	92
SHOW STATEMENT	94
START STATEMENT	97
TRY STATEMENT	99
UPDATE STATEMENT	101
WHILE STATEMENT	102
XEQ STATEMENT	103
! STATEMENT	105
* STATEMENT	106
<b>CHAPTER FOUR: FILE TYPES</b>	<b>107</b>
ALLBASE FILE TYPE	109
ARCHIVE FILE TYPE	113
CSV FILE TYPE	119
DB2 FILE TYPE	131
FIXED FILE TYPE	137

---

---

IMAGE FILE TYPE .....	147
ODBC FILE TYPE.....	163
ORACLE FILE TYPE.....	176
REMOTE FILE TYPE.....	183
REPORT FILE TYPE.....	196
TEXT FILE TYPE.....	199
TEXT FILE TYPE.....	206
<b>CHAPTER FIVE: WAREHOUSE EXPRESSIONS.....</b>	<b>213</b>
IDENTIFIERS.....	215
CONSTANTS .....	218
NUMERIC CONSTANTS.....	218
STRING CONSTANTS .....	218
SYSTEM CONSTANTS .....	219
\$ERR SYSTEM VARIABLE .....	220
OPERATORS.....	224
DATE OPERATORS .....	232
NULL OPERATIONS .....	234
BUILT-IN FUNCTIONS.....	236
<b>CHAPTER SIX: DATA TYPES.....</b>	<b>273</b>
ALLBASE DATA TYPES.....	275
IMAGE DATA TYPES .....	288
ODBC DATA TYPES.....	304
ORACLE DATA TYPES .....	326
SQL DATA TYPES .....	342
WAREHOUSE DATA TYPES .....	359
<b>CHAPTER SEVEN: INSTALLATION AND EXECUTION.....</b>	<b>373</b>
WAREHOUSE CLIENT AND SERVER OPTIONS.....	375
INSTALLATION ON MPE/IX.....	391
MPE/IX INSTALLATION CONSIDERATIONS .....	392
RUNNING WAREHOUSE ON MPE/IX .....	392
RUNNING THE WAREHOUSE SERVER ON MPE/IX.....	393
INSTALLATION ON UNIX/LINUX .....	396
UNIX INSTALLATION CONSIDERATIONS .....	397
RUNNING WAREHOUSE ON UNIX .....	397
RUNNING THE WAREHOUSE SERVER ON UNIX .....	398
WINDOWS INSTALLATION CONSIDERATIONS .....	402
RUNNING WAREHOUSE ON WINDOWS .....	402
RUNNING THE WAREHOUSE SERVER ON WINDOWS.....	402
UNINSTALLING WAREHOUSE ON WINDOWS.....	407
VALIDATING WAREHOUSE.....	409
AUTHORIZING WAREHOUSE SERVER ACCESS .....	410
CHECKING WAREHOUSE SERVER CONNECTIONS.....	417
ENVIRONMENT VARIABLES .....	420
<b>APPENDIX A: TRANSACTIONS AND ERROR HANDLING.....</b>	<b>423</b>
TRANSACTIONS .....	424
ERROR HANDLING.....	427
<b>APPENDIX B: PREPROCESSING .....</b>	<b>429</b>
PREPROCESSING.....	430

---

---

<b>APPENDIX C: CHARACTER MAPS .....</b>	<b>439</b>
CHARACTER MAP FILES .....	440

---



# **Chapter One**

## **Introduction**

### Product Overview

Warehouse is a general utility that allows you to select and move data. Warehouse is not sensitive to where the data is coming from or going to. It handles all of the work involved in moving data (data type conversions, data placement) behind the scenes.

Warehouse is used to move data:

- When migrating your data from one database/platform to another
- Into a data warehouse
- Archiving stale data offline or to a historical database
- Retrieving archived data: Back into its original database, into an entirely different database, or directly into a report
- Creating test environments

Warehouse employs advanced client/server technology to move your data. You write a simple Warehouse script that accesses either local or remote databases. When the script is executed, your data moves exactly where you want it, even to a distant system running a completely different operating system and database.

### How to Use This Manual

The *Warehouse Reference Manual* is designed to meet the needs of on-line and batch users. The *Warehouse Reference Manual* is intended to be a complete functional description of the Warehouse product. In Chapter Two, Warehouse Scripts, the components of scripts and archiving concepts are introduced through examples. Chapter Three, Warehouse Statements, provides detailed information about each Warehouse statement, including syntax, defaults, restrictions, and examples. Chapter Four, File Types, describes the use of each Warehouse statement with all of the supported file types. Chapter Five, Warehouse Expressions, explains the use of Warehouse



## Introduction

---

expressions including identifiers, operators and built-in functions. [Chapter Six, Data Types](#), explains the details of each of the data types supported by Warehouse. [Chapter Seven, Installation and Execution](#), explains the details of how Warehouse is installed and run on each of the supported platforms.

### **More Information on the Web**

We have extensive help available through the following on-line pages and downloads.

#### Warehouse User Guide

This user guide details real world problems and provides example Warehouse script solutions. The Warehouse User Guide can be found by:

1. Navigate to <http://www.taurus.com>
2. Click on Downloads under the Support section
3. Click on Software and Documentation downloads
4. Click on Warehouse User Guide (.pdf format).

Access to this site requires a Logon ID and a Password that can be obtained by emailing [support@taurus.com](mailto:support@taurus.com)

#### Frequently Asked Questions (FAQ)

The FAQ file can be found at <http://www.taurus.com/support/faq.htm>. It is updated on a regular basis and is an external solution database. In addition to documenting functionality of our Software, many answers to problems can be found here, with step by step solution instructions.

#### Self Service Knowledge Base

This is a Taurus Client resource that provides access to our internal solution database. Client users can create trouble tickets directly in our case tracking system, upload related files and view ticket status. The Knowledge Base system can be found at:

<http://www.taurus.com/support/selfserve.htm>

Users are required to have a Logon ID and Password which they can obtain by emailing [support@taurus.com](mailto:support@taurus.com)

### Product Version

If you place a customer support call, it is important that you know the Warehouse version number that you are using. Since changes are made to Warehouse between releases, the Customer Support Representative may not be able to assist you properly without knowing the product version you are using.

The version number appears when Warehouse is run, and has four parts: major release (single character), update (number), fix level (numbers), and operating system code. Warehouse 2.01.0010-M is an example of a product version number with a major release of 2, update level of 01, fix level of 0010, and operating system of M, which refers to MPE/iX. The operating system codes are as follows:

- A Digital Equipment Corp. DEC Unix Operating system on Alpha architecture
- C Santa Cruz Operations OpenServer Operating system
- F Hewlett-Packard HP-UX PA RISC 2.0 (Oracle dynamically linked)
- G Hewlett-Packard HP-UX Itanium (Oracle dynamically linked)
- H Hewlett-Packard 9000 HP-UX PA RISC 1.1 (Oracle statically linked)
- I Silicon Graphics IRIX Operating system
- J SunOS 5.10 Solaris x86
- L Red Hat Linux
- M Hewlett-Packard 3000 MPE/iX Operating system.
- Q IBM AS/400 system iSeries
- R IBM RS6000 system running AIX
- S Sun Microsystems system running SunOS/Solaris SPARC
- T Debian Linux (64-bit)
- U Microsoft Windows x64 (64-bit)
- V DEC VMS (No Longer Supported)
- W Microsoft Windows/Windows NT Operating system (32-bit)
- X Unknown

Knowing the correct version can save you time when calling Customer Support.

### Contact Us

Phone: (650) 482-2022 x2  
Monday through Friday  
8:00 a.m. to 5:00 p.m. Pacific Time  
(holidays excluded).

FAX: (650) 482-2010  
Please include name, company name, and  
phone number.

Email: support@taurus.com  
Monday through Friday  
6:00 a.m. to 6:00 p.m. Pacific Time  
(holidays excluded).

### On-line Help

Web FAQ  
[www.taurus.com/support/faq.htm](http://www.taurus.com/support/faq.htm)

Self Service Portal:  
[www.taurus.com/support/selfserve.htm](http://www.taurus.com/support/selfserve.htm)

### Triage

Cases received by technical support are triaged so that production stopping issues are handled first. If you have a production stopping issue, please email supporting documentation and call us. Non emergency inquiries are handled in the order they are received.

**Chapter Two**  
**Warehouse Scripts**

### Scripting Language Overview

Warehouse uses a simple scripting language to define each operation. Usually, Warehouse scripts are initially created using Taurus Software's graphical user interface (GUI) DataBridger Studio. After scripts have been created using DataBridger Studio, they may be enhanced using any text editor.

The first part of a script contains `OPEN` statements that specify which databases and files are to be accessed by the script. The second part usually sets up script variables using the `DEFINE` statement. Following the `OPENS` and `DEFINES`, data is read from the source using the `READ` statement and written to the target using the `COPY` statement. The "read loop" is the primary construct in a Warehouse script. The `READ` statement specifies which table, dataset, or file is to be read and which records are to be selected. The `READ` loop is terminated with an `ENDREAD` statement. Between the `READ` and `ENDREAD` all statements are executed for each record selected. `READ` statements may be nested to access several sources simultaneously. Warehouse scripts may contain other statements to delete and update data, and language constructs such as `IF` and `WHILE`.

### Sample Script

The following is a sample Warehouse script that moves customer records from an `IMAGE` database on an HP3000 to an Oracle database on an HP-UX system:

```
1.      * Loads CUST data into Oracle
2.      OPEN CUST IMAGE CUSTDB.DATA &
3.          PASS=PW MODE=1
4.      OPEN CUSTTGT REMOTE UXSYS &
5.          USER=whux PASS=whuxpwd &
6.          ORACLE scott/tiger &
7.          SID=sid04 HOME=/u01/oradata/ora

8.      DEFINE CTGT : USING CUSTTGT.CUST

9.      READ CSRC = CUST.CUST-MASTER &
10.         FOR ACTIVE = "Y"
```

```
11.      SETVAR CTGT.CUSTOMER_ID = &
12.      CSRC.CUSTOMER-ID
13.      SETVAR CTGT.CUSTOMER_NAME = &
14.      CSRC.CUSTOMER-NAME
15.      SETVAR CTGT.ADDRESS = &
16.      CSRC.ADDRESS
17.      SETVAR CTGT.PHONE = &
18.      CSRC.PHONE
19.      COPY CTGT TO CUSTTGT.CUST
20.  ENDREAD
21.  GO
```

Line 1: Comment line explaining what the script does. All lines that begin with an asterisk (\*) are considered comment lines.

Line 2: Opens the IMAGE database called CUSTDB.DATA. (IMAGE refers to the database system found on Hewlett-Packard HP3000 MPE computer systems. IMAGE is also called TurboIMAGE and IMAGE/SQL.) The database is assigned a database tag of CUST that is used to reference the database in the remainder of the script. Warehouse is case insensitive so the OPEN statement could be been open or Open.

The line ends in an ampersand (&) indicating that the OPEN statement is to be continued on the next line.

Line 3: This is a continuation of line 2. The IMAGE database CUSTDB.DATA is opened using a password PW and a database open mode of 1, which allows read/write access.

Line 4: Opens the Oracle database on a remote system named UXSYS. The database type of REMOTE is used to indicate a remote system running a Warehouse server. In this case, the remote system is an HP-UX

system running Oracle. The database is assigned a database tag of `CUSTTGT` that is used to reference the Oracle database in the remainder of the script.

Line 5: This is a continuation of line 4. A user name of `whux` and a password of `whuxpwd` is used to connect to the remote system. For this connection to work, it must have been previously authorized by the system administrator of the Unix system `UXSYS`. In this example, a plain text password is used. Warehouse supports the use of encrypted passwords so that plain text passwords are not necessary.

Line 6: This is a continuation of line 4. Indicates the remote database type is an Oracle database to be accessed with a user name of `scott` and a password of `tiger`. It is possible to use encrypted passwords so that plain text passwords are unnecessary.

Line 7: This is a continuation of line 4. The Oracle SID and Oracle home necessary to open the database are specified. This concludes the `OPEN` statement begun in line 4.

Warehouse is designed to be insensitive to the physical location of the databases. With the exception of the `OPEN` statements, a script will run the same regardless of which system it executes. To run this script on the Unix system, simply change line 2 to open a remote IMAGE database and line 4 to open a local Oracle database. No other changes are necessary in the remainder of the script.

Line 8: Defines a record variable named `CT` that will be copied to the Oracle database later



in the script. CT is defined to be a record with a layout exactly like the table CUST within the CUSTTGT (Oracle) database.

Line 9: The READ statement begins a Warehouse "read loop." A read loop is a READ statement, followed by the body of the read loop, followed by a matching ENDREAD statement. The body of the read loop is executed once for every record selected by the read statement. In this case, the CUST-MASTER file is read from the CUSTDB IMAGE database. All statements between the READ statement and the ENDREAD statement on line 20 are executed for each CUST-MASTER record selected.

The records read from CUST-MASTER are assigned a read tag of CSRC. The records are accessed within the script using CSRC. Fields within CUST-MASTER are accessed using the construct CSRC.fieldname.

The line ends in an ampersand (&) indicating that the READ statement is to be continued on the next line.

Line 10: This line is a continuation of line 9 and is considered part of the READ statement. The FOR keyword is used to specify which records the READ statement is to select. In this case, all records having a value of Y in the ACTIVE field of the CUST-MASTER file are selected.

Line 11: Sets the value of the field CUSTOMER\_ID within the variable CT defined on line 8. The (&) continues the statement to the next line.

Line 12: The value CUSTOMER-ID from the CSRC

record, which was read from the CUST-MASTER dataset read in line 9. Lines 11 through 18 simply load the record variable CT with values read from CUST-MASTER.

- Line 13: Sets the value of the field CUSTOMER\_NAME within the variable CT. The (&) continues the statement to the next line.
- Line 14: The value CUSTOMER-NAME from the CSRC record.
- Line 15: Sets the value of the field ADDRESS within the variable CT. The (&) continues the statement to the next line.
- Line 16: The value ADDRESS from the CSRC record.
- Line 17: Sets the value of the field PHONE within the variable CT. The (&) continues the statement to the next line.
- Line 18: The value PHONE from the CSRC record.
- Line 19: Copies (inserts or writes) the CTGT record variable to the CUST table in the target Oracle database.
- Line 20: Terminates the CUST-MASTER read loop begun in line 4.
- Line 21: Terminates script definition and begins script processing. No records are read or written until the GO statement. Statements before GO are simply checked for syntax and compiled internally. The GO statement begins script execution.

**Chapter Three**  
**Warehouse Statements**

### Chapter Overview

This chapter describes in detail each of the statements that may be used in a Warehouse script. The function of each statement is discussed, along with its syntax, considerations and examples.

### Warehouse Scripts

Warehouse scripts are contained in a text file and may be edited with any text editor. Statements are terminated by the end of line. Blank lines are ignored.

White space characters (spaces and tabs) at the beginning of a line and multiple white space characters are ignored.

All statements, keywords, and identifiers are case insensitive, converting everything to uppercase before comparison. To treat an identifier as case sensitive, enclose the identifier in curly braces ( { } ).

#### Example 1

```
DEFINE tablename : ODBC CHAR(10)
SETVAR TABLENAME = "Dining"
```

Warehouse treats both case-versions of `tablename` and `TABLENAME` the same. After execution, `tablename` holds the value "Dining".

#### Example 2

```
OPEN rec-in ODBC InputTable USER={foo}
READ r = rec-in.{dbo}.table1
```

Curly braces are required on the User ID and the table schema to keep them lower case for comparison. Without the curly braces, the user id and schema would not be found.

### Line Continuation

To continue a statement onto the next line an ampersand (&) is used. There is no maximum line length.

### Comments

Any line where the first nonwhite-space character is an asterisk (\*) is considered a comment. Comments may also be placed at the end of a line using two slashes (/ /) to denote the end of line and beginning of the comment. e.g.

```
ENDREAD // CUSTOMER Table READ
```

The .csv mapping files for DataBridger Studio contain comment lines beginning with two slashes (/ /). When these mapping files are processed, Studio substitutes an asterisk for the double slashes in the script output.

**CALL****Calls a User-Defined Function.**

The `CALL` statement is used to call a user-defined function that does not return a value. Call is also used to call the Warehouse `DIRECT` function.

**Syntax**

```
CALL function-name [( function-parms )]
```

`function-name` is the name of the user-defined function to be called. See the `FUNCTION` statement in this chapter for information on user-defined functions.

`function-parms` is a list of function parameters. The parameters must match those of the function definition. See the `FUNCTION` statement in this chapter for information on user-defined functions.

**Considerations**

The `CALL` statement is only used to call functions that do not have a return value. To call a function with a return value, use the function name in an expression.

**Examples****Example 1**

```
FUNCTION PRINT_PART(PN : ORACLE CHAR(8))
```

```
    READ P = PTDB.PARTS FOR PARTNO = PN
        PRINT PARTNO, PARTDESC
    ENDREAD
ENDFUNCTION
```

```
READ OD = ODB.ORDLINES &
    FOR ORDNO = "109152"
    CALL PRINT_PART(ORDPARD)
ENDREAD
```

A function called `PRINT_PART` is defined that has one formal parameter `PN` of type `ORACLE CHAR(8)`. This function reads from the `PARTS` table in the database opened with a tag of `PTDB`. Records where `PARTNO` equals the formal parameter `PN` are selected and the fields `PARTNO` and `PARTDESC` are printed. The function `PRINT_PART` simply prints the part number and part description for a given

part number.

To use the function, the table `ORDLINES` is read from the database opened as `ODB`, where `ORDNO` equals "109152". After reading from `ORDLINES`, the function `PRINT_PART` is called using `ORDPARD` as an actual parameter.

See the `FUNCTION` statement in this chapter for more examples of the `CALL` statement.

### Example 2

```
FUNCTION PRINT_RETAIL( &
    PCOST : ORACLE NUMBER, &
    PQTY  : ORACLE NUMBER )

    PRINT PCOST, PQTY, PCOST * PQTY
ENDFUNCTION

READ OD = ODB.ORDLINES &
    FOR ORDNO = "123456"
    CALL PRINT_RETAIL(ORDCST, ORDQTY)
ENDREAD
```

This example shows a function that takes two parameters and prints their value and their product. The function is called from a read loop that passes all the lines of Order number 123456.

**CLOSE****Causes Immediate Close of Database.**

The `CLOSE` statement causes an immediate close of a database.

**Syntax**

`CLOSE db-tag`

`db-tag` is the database tag of the database/file to be closed.

**Considerations**

The `CLOSE` statement immediately closes a database. Once a database has been closed it may not be accessed in the remainder of the script or during script processing.

The most common use of `CLOSE` is when a database has been opened only to obtain the layout of tables or datasets. After `FORMAT` statements referencing the database have completed, the database is closed.

**Examples****Example 1**

```
OPEN CDB IMAGE CUSTDB PASS=IMPASS MODE=5
FORMAT CFMT : USING CDB.CUSTS
CLOSE CDB
OPEN CFIL FIXED CUSTFIL
READ CDATA = CFIL FORMAT CFMT
  PRINT CUSTNO, CUSTNAME
ENDREAD
```

This example opens an Image database `CUSTDB` using a database tag of `CDB` to reference the database in the remainder of the script. A `FORMAT` statement is then issued that defines a record layout called `CFMT` that is identical to the record layout of the `CUSTS` dataset within the `CUSTDB` database. Once the `FORMAT` statement has been issued, `CUSTDB` is no longer needed in the script, so a `CLOSE` statement is used to close the database. A fixed length record file called `CUSTFIL` is then opened using a database tag of `CFIL`. The records in `CUSTFIL` have the same format as the records from `CUSTS` within `CUSTDB` so they are read using the format `CFMT`. The fields from the file `CUSTNO`



and CUSTNAME fields are then printed.

**COMMIT****Causes a Warehouse commit operation.**

The COMMIT statement causes a Warehouse commit operation. The COMMIT statement causes all databases that have been accessed during the transaction to be committed.

**Syntax**`COMMIT`**Considerations**

In a typical script the COMMIT statement is unnecessary. The COMMIT statement is only provided for sophisticated transaction management and may cause problems if used inappropriately.

The COMMIT statement performs a commit operation on all databases accessed by the script since the previous commit operation. There is no way to commit a single database.

The effect of a COMMIT statement depends on the type of database accessed. The effect is as follows for each database type:

ALLBASE	Performs database commit
ARCHIVE	No effect
CSV	No effect
DB2	Performs database commit
FIXED	No effect, unless MPE/iX message file opened with NDR, in which case most recent record read is removed.
IMAGE	Unlocks database if locked and locking not MANUAL. Calls DBXEND if locking is ROLLBACK.
ODBC	Performs database commit
ORACLE	Performs database commit
REMOTE	Depends on underlying database
REPORT	No effect
TEXT	No effect

**Examples****Example 1**

```
OPEN RI &  
  REMOTE MPESYS USER=MGR.DBMGR &  
  IMAGE IMGDB PASS=IMPASS MODE=5
```

```
OPEN LO ORACLE SCOTT/TIGER
DEFINE COUNTER : INTEGER
READ CUST = RI.CUSTOMERS
  COPY CUST TO LO.CUSTOMERS
  SETVAR COUNTER = 0
  READ CT = RI.CUST-TRANS &
    CUSTNO = CUST.CUSTNO
  IF COUNTER = 100
    COMMIT
    SETVAR COUNTER = 0
  ENDIF
  COPY CT TO LO.CUST_TRANS
  SETVAR COUNTER = COUNTER + 1
ENDREAD
ENDREAD
```

This example opens a remote Image database IMGDB on the MPE/iX system MPESYS. It then opens a local Oracle database. An integer variable called COUNTER is defined. The CUSTOMERS dataset is read from the remote Image database and copied to the local Oracle database. COUNTER is set to zero and then matching CUST-TRANS records are read from the Image database. If 100 transaction records have already been read and copied to the Oracle database since the last commit, a COMMIT statement is executed and COUNTER is reset to zero. This permits a limit on the number of CUST-TRANS records copied within a database transaction. The CUST-TRANS record is then copied to the Oracle database and COUNTER is incremented.

### Example 2

```
OPEN mast ODBC DAILYWORK
OPEN emp ODBC EMPMAST
OPEN phon ODBC PHONEMAST
OPEN ship ODBC SHIPPINGMAST

READ r = mast.EMPINFO
  TRY
    READ e = emp.EMP &
      FOR EMPNO = r.EMPNO
      UPDATE e SET FNAME = r.FNAME
    ENDREAD

    READ p = phon.CONTACTS &
```

```
        FOR EMPNO = r.EMPNO
        UPDATE p SET HOMEPEH = r.EMPPHONE
    ENDREAD

    READ s = ship.LOCATIONS &
    FOR EMPNO = r.EMPNO
    UPDATE s SET HOMEADDR = r.SHIPTO
    ENDREAD

    UPDATE r SET STATUS = "PROCESSED"
    COMMIT
RECOVER
    ROLLBACK
    PRINT r.EMPNO, ": unable to process"
ENDTRY
ENDREAD
```

This example shows the combined use of a `COMMIT` and `ROLLBACK`. The input table `EMPINFO` contains update information used to populate data to three other tables. After all three tables are updated, the input record's status is changed to denote that it's been processed, and a `COMMIT` is issued to update all the database changes. The `TRY RECOVER` block is used to prevent partial updates if errors are encountered before all the tables are updated. If an error occurs the `RECOVER` block issues a `ROLLBACK` command and prints a message to the standard output that there was a problem with processing the input record. The `ROLLBACK` will reverse any table updates since the last `COMMIT`.

**COPY****Copies a Record to a File.**

The `COPY` statement copies (writes) a record to a file. The source record may be either a read tag created with a `READ` statement or a record created with the `DEFINE` statement. If the source record has a different format than the output file, Warehouse automatically converts the source record to the format of the output file.

**Syntax**

```
COPY record TO output-file
    [FORMAT format-name]
    [[;] ERRORS TO error-file]
    [[;] WAIT | NOWAIT]
```

`record` is the name of a record created with either the `DEFINE` statement or a read tag created by the `READ` statement. See the `READ` statement in this chapter for information on read tags.

`output-file` is the name of the file to which the record is copied. When copying to a database, `output-file` is in the format `db-tag.table-name`. When copying to a CSV, `FIXED` or `TEXT` file, `output-file` is simply a `db-tag`. See [Chapter Four, File Types](#), for more information.

`format-name` is the name of a format previously created with the `FORMAT` statement. When `format-name` is specified, the format of the `output-file` is redefined to be that of the format specified by `format-name`.

`error-file` is the name of the file to which the record is copied in the event an error occurs when copying to `output-file`. When copying error records to a database or archive file, `error-file` is in the format `db-tag.table-name`. To copy to a CSV, `FIXED` or `TEXT` file in the event of an error, `error-file` is simply the `db-tag` of the file. See [Chapter Four, File Types](#), for more information.

`NOWAIT` only has an effect when copying to a

remote database. `NOWAIT` causes Warehouse to continue processing the script without waiting for the record to actually be copied to the database. This allows Warehouse to continue processing the script without waiting for a response from the server. `NOWAIT` is default when copying to a remote database.

**Warning:** Using `NOWAIT` (the default) can have error recovery implications when used inside a `TRY/RECOVER` block, because if an error occurs the `RECOVER` statement will NOT be entered with the record that had the error.

`WAIT` only has an effect when copying to a remote database. `WAIT` causes Warehouse to wait for the server to actually copy the record to the database before continuing. `WAIT` is typically used to enhance error recovery within a `TRY` block.

Using `WAIT` can have significant performance implications since the Warehouse client must wait for a response from the server for each record written.

## Considerations

If `record` has a different format than the `output-file`, Warehouse automatically converts the record to the destination format as follows:

- Different order: Warehouse automatically copies each field from `record` to the field of the *same name* in the `output-file`.
- New fields in the `output-file`: If the `output-file` has additional items, Warehouse initializes numeric type items to zeroes and string type items to spaces. For a full list of item types, see [Chapter Six, Data Types](#).
- Missing fields in the `output-file`: Warehouse does not copy fields in `record` that do not have a corresponding field name in the `output-file`.

- Different data types: Fields from `record` that have a different data type than fields of the same name in the `output-file` are automatically converted to the data type of the field in the `output-file`.
- Data length changes in alphanumeric fields: Warehouse truncates data when moved to a smaller field. Warehouse pads with spaces when moved to a larger field.

If `ERRORS TO` is specified, Warehouse first attempts to copy the record to `output-file`. If an error is encountered, an attempt is made to copy the record to `error-file`. If an error occurs when copying to `error-file`, an error condition is caused.

If no `error-file` is specified, and an error is encountered during the copy, an error condition is caused. An error condition inside a `TRY` statement causes control to switch to the corresponding `RECOVER` statement. An error condition outside a `TRY` statement causes script processing to stop.

## Examples

### Example 1

```
OPEN DB1 IMAGE DBONE
OPEN DB2 IMAGE DBTWO.GRP.ACCT
CREATE AR ARCHIVE ARCHFILE
READ SET-1 = DB1.SET-1
    COPY SET-1 TO AR.DBONE.SET-1
ENDREAD
READ SET-2 = DB2.SET-2
    COPY SET-2 TO AR.DBTWO.SET-2
ENDREAD
```

This example reads and copies all records from the file `SET-1` file in the `IMAGE` database `DBONE` to the archive file `ARCHFILE`. Then, it copies all of `SET-2` in the `IMAGE` database `DBTWO.GRP.ACCT` to the archive file.

Example 2

A database was archived many months ago that had the following structure for the CUSTOMER file:

CUST-NO,	Z10
CUST-NAME,	X40
CUST-ADDR1,	X40
CUST-ADDR2,	X40
LASTSO,	Z6
STATUS,	X8

The structure of the CUSTOMER file has since been changed to the following:

CUST-NO,	Z10
CUST-NAME,	X40
CUST-ADDR1,	X40
CUST-ADDR2,	X40
CUST-ADDR3,	X40
LASTSO,	X10

Note that CUST-ADDR3 has been added, STATUS has been deleted, and the data type and size of LASTSO has been changed.

The script used to retrieve CUSTOMER from the archive file is as follows:

```
OPEN AR ARCHIVE ARCHFILE
OPEN CUST IMAGE CUSTDB
READ C = AR.CUSTOMER
  COPY C TO CUST.CUSTOMER
ENDREAD
```

Warehouse copies the contents from CUST-NO, CUST-NAME, CUST-ADDR1, and CUST-ADDR2 on the archive tape to the corresponding fields in CUSTOMER. The STATUS field is not copied because it doesn't exist in the target file. The CUST-ADDR3 field is initialized to all spaces and the data in the LASTSO field is converted from Z6 format to X10 format. The data is left-justified and the leading zeroes are removed.



Example 3

```
OPEN DB1 IMAGE DBONE
OPEN DB2 REMOTE GREEN &
    USER=MYUSER PASS=MYPASS &
    ORACLE SCOTT/TIGER SID=ORCL
CREATE EF ARCHIVE ERRFILE
READ CUST = DB1.CUSTOMERS
    COPY CUST TO DB2.CUSTLIST &
    ERRORS TO EF.BADCUSTS
ENDREAD
READ ORDS = DB1.ORDMAST
    COPY ORDS TO DB2.ORDERS &
    ERRORS TO EF.BADORDS
ENDREAD
```

This example opens the IMAGE database DBONE on the local system and assigns it a tag of DB1. It then opens an Oracle database on the remote system GREEN. A user name of MYUSER and a password of MYPASS is used to establish the connection on GREEN. An Oracle user name of SCOTT and Oracle password TIGER is used along with a system id (SID) of ORCL. The Oracle database is assigned a tag of DB2. An archive file called ERRFILE is created and assigned a tag of EF.

Once the files have been opened, CUSTOMERS records are read from DBONE and copied across the network into the table CUSTLIST in the Oracle database on GREEN. Any records that cannot be copied into CUSTLIST in the Oracle database are instead written to the table BADCUSTS within the archive file ERRFILE. After the customer records are read, order records are read from ORDMAST and copied to ORDERS. Any records that cannot be copied are written to the table BADORDS within the archive file ERRFILE.

Example 4

A school has an ORACLE table named CONTACTS with the following structure:

ID	NUMBER ( 8 )
----	--------------

NAME	CHAR(40)	ALLOW NULLS
PHONE	NUMBER(11)	ALLOW NULLS
EMAIL	CHAR(30)	ALLOW NULLS
LOCATION	CHAR(20)	ALLOW NULLS
TXNDATE	DATE	ALLOW NULLS

A group of parents have a call tree database with a SQL Server table named CALLIST with the following structure:

ID	INT	
NAME	CHAR(25)	ALLOW NULLS
ADDR	VARCHAR(50)	ALLOW NULLS
PHONE	DECIMAL(15,0)	ALLOW NULLS
Email	VARCHAR(50)	ALLOW NULLS
TXNDATE	DATETIME	ALLOW NULLS

Notice one of the column names has mixed case. We want to create a script that copies records from the ORACLE table CONTACTS to the SQL Server table CALLIST. MyPTA is set up as an ODBC connection to the SQL Server database.

```
OPEN SCHOIN ORACLE scott/tiger
OPEN PTAOUT ODBC MyPTA
READ r = SCHOIN.CONTACTS
      COPY r TO PTAOUT.{dbo}.CALLIST
ENDREAD
```

The COPY command in Warehouse will automatically convert the data between the two systems by exactly matching the Column Names with the following results:

ID is converted from NUMBER(8) to INTEGER;  
NAME is truncated from 40 to 25 characters because the target field is smaller than the source field;  
ADDR is null in the target table because the column doesn't exist in the source table;  
PHONE is converted from NUMBER(11) to DECIMAL(15,0);  
EMAIL is null in the target table because the column name is a different case which causes it to not match up to the source column;  
TXNDATE has the date portion copied with zeroes in the time portion of the target field because the

source contains only date information.

**CREATE****Creates and opens a new file.**

The `CREATE` statement creates a new archive file or file to be accessed within the Warehouse script.

**Syntax**

```
CREATE db-tag file-type [file-parms]
```

`db-tag` is the name of the database tag used to reference the database or file in the remainder of the script. A database tag is also called a *file tag* when used to access a simple file instead of a database.

`file-type` is the type of file to be created. Supported values of `file-type` are:

ARCHIVE	Warehouse archive file
CSV	Comma separated file
FIXED	Fixed record length file
REMOTE	Remote access file
TEXT	Text (character) file

`file-parms` is the file name of the file to be created and any `file-type` specific parameters needed to create the file. The exact meaning of `file-parms` depends on the `file-type`. See [Chapter Four, File Types](#), for more information.

**Considerations**

The `OPEN` and `CREATE` statements perform the same basic function, i.e. make a file or database available for access by Warehouse. The difference is that `OPEN` opens an existing database or file and `CREATE` makes a new one.

`CREATE` and `OPEN` statements are generally placed at the top of your script file because files must be opened before they may be referenced within the script.

The `CREATE` statement overwrites any existing file.

**Examples**Example 1

```
CREATE OUT FIXED TMPFILE &
```

```
MODE=WRITE ERASE
```

This example opens the fixed length record binary file `TMPCFILE` using mode `w` for write access. The file tag `OUT` is used to access this file in the remainder of the script.

### Example 2

```
CREATE AF ARCHIVE ARCHFILE
```

This example creates `ARCHFILE` as a new archive file for output.

### Example 3

```
CREATE RT REMOTE GRAPE.TAURUS.COM &  
  USER=mary PASS=mpw &  
  TEXT /users/mary/wh/scriptlog
```

This example creates a text file `scriptlog` in the directory `/users/mary/wh/` on the remote Unix computer `GRAPE.TAURUS.COM`. The remote user is logged on as user `mary` using the password `mpw`.

**DEFINE****Defines variables for use in a script.**

The `DEFINE` statement establishes an internal variable and defines its data type for later use in a script. Variable values are assigned using the `SETVAR` statement. If the `DEFINE` statement is within a function definition, the `DEFINE` statement establishes a local variable for use only within the function.

**Syntax**

```
DEFINE var-name [, var-name][,...] :  
    data-type  
    [VALUE initial-value]  
    [ALLOW NULLS]  
    [CHARSET "character set name"]
```

where data-type is:

```
[db-type] simple-type  
FORMAT format-name  
RECORD item-list END  
USING file
```

`var-name` is the name the variable being defined. Several variables of the same type may be defined by separating the names by commas.

`db-type` specifies the database originator of the type. Examples of `db-type` are `ALLBASE`, `IMAGE`, and `ORACLE`. See [Chapter Six, Data Types](#), for details on specifying data types.

`simple-type` specifies the data type of the `var-name`. See [Chapter Six, Data Types](#), for details on specifying data types.

`FORMAT format-name` specifies that the variable being defined be given the format of a previously created format using the `FORMAT` statement.

`RECORD item-list` specifies the definition of a record. See [Chapter Six, Data Types](#), for details on specifying the items within records.

`USING file` specifies that the variable being

defined has the same format as a file previously opened with the `OPEN` statement. This is typically in the format of `db-tag.table-name`.

`initial-value` specifies an initial value for the variable being defined. If more than one variable is defined, all variables receive the same initial value. The `initial-value` must be a constant expression, i.e. it may not reference other variables.

`ALLOW NULLS` specifies this data type may contain no value. When specifying both a `CHARSET` and `ALLOW NULLS`, the keywords may appear in either order.

`CHARSET "characterSetName"` associates a string data type with a character set. The character set name, `characterSetName`, must be enclosed in quotation marks. When specifying both a `CHARSET` and `ALLOW NULLS`, the keywords may appear in either order.

When operations with strings of differing character sets are performed, an automatic character set conversion is done using the `CMAP` function. The operations that can generate an automatic `CMAP` are:

Comparison operators: `<`, `<=`, `=`, `>=`, `>`,  
`<>`, `==`

String assignment: `SETVAR` statement, `UPDATE` statement

String concatenation: `||`

Strings may or may not have a character set. When a string operation is performed, no character set conversion is done if either string has no character set or if the strings have the same character set. Conversion is only done when both strings have a character set and the two character sets differ.

Automatic character set conversion may be overridden using the `CONVERT` or `FIELD` functions using a target type with no character set or a

different character set.

For example, the following two code snippets result in the same value in TGTNAM:

Snippet 1:

```
DEFINE SRCNAM : IMAGE X8
DEFINE TGTNAM : ODBC CHAR(8)
SETVAR TGTNAM = CMAP(SRCNAM, &
    "HP-ROMAN8", "ISO8859-1")
```

Snippet 2:

```
DEFINE SRCNAM : IMAGE X8 CHARSET &
    "HP-ROMAN8"
DEFINE TGTNAM : ODBC CHAR(8) CHARSET &
    "ISO8859-1"
SETVAR TGTNAM = SRCNAM // Auto CMAP here
```

## Considerations

The `DEFINE` statement can establish either global or local variables. If the `DEFINE` statement is outside a `FUNCTION` definition, *global* variables are established, which means the variable can be accessed anywhere in the script. If the `DEFINE` statement is inside a `FUNCTION` definition, *local* variables are established which may only be accessed only within the function.

If no `initial-value` is specified, the variables are initialized depending upon the `data-type` family as follows:

<u>Data Type Family</u>	<u>Initial Value</u>
Date	January 1, 1901
Datetime	01-JAN-1901 00:00:00
Fixed length Binary	Binary zeroes
Fixed length Character	Spaces
Interval	0 00:00:00
Logical	False
Numeric	Zero
Record	Initializes fields
Time	12:00:00 AM
Variable length	Set length to zero

## Examples

### Example 1



```
DEFINE NUM'RECORDS : IMAGE I2
```

Define a variable called NUM'RECORDS as an IMAGE 4-byte signed integer. NUM'RECORDS is initialized to zero.

### Example 2

```
DEFINE AMOUNT : ODBC DECIMAL(6,2) &  
VALUE -9999.99
```

Defines the variable AMOUNT as an ODBC number with 6 digits of precision: 4 to the left of the decimal point, and 2 digits to the right. AMOUNT is given an initial value of -9999.99.

### Example 3

```
DEFINE COMPANY-NAME : ORACLE CHAR(20)
```

Defines the variable COMPANY-NAME as Oracle character data with a length of 20 bytes.

### Example 4

```
DEFINE COMPANY : RECORD  
    CO-NAME      : ORACLE CHAR(40)  
    CO-NUM       : ORACLE CHAR(10)  
    STATUS       : ORACLE CHAR(8)  
END
```

Defines a 58 byte record variable COMPANY that has three Oracle character elements: CO-NAME, CO-NUM, and STATUS. Each field within COMPANY is initialized to spaces.

### Example 5

```
OPEN DB IMAGE MYDB PASS=MYPASS  
DEFINE COMPANY-REC : USING DB.COMPANY
```

Defines a record called COMPANY-REC that has the exact same format as the dataset COMPANY within the database MYDB.

## Example 6

```

FORMAT COMPANY-REC : RECORD
    CO_NAME      : ORACLE CHAR(40)
    CO_NUM       : ORACLE CHAR(10)
    STATUS       : ORACLE CHAR(8)
END
DEFINE COMPANY-REC2 : FORMAT COMPANY-REC

```

Defines a record called COMPANY-REC2 that has the format of COMPANY-REC.

## Example 7

```

FUNCTION COPY_ORD &
    (VAR ORDREC : USING DB.ORDERS)

    DEFINE OR : RECORD
        ORDNO    : ORACLE CHAR(8)
        CUSTNO    : ORACLE CHAR(8)
        ORDDATE   : ORACLE DATE
    END

    SETVAR OR.ORDNO    = ORDREC.ORDNO
    SETVAR OR.CUSTNO    = ORDREC.CUSTNO
    SETVAR OR.ORDDATE   = ORDREC.ORDDATE
    COPY OR TO ORDFILE

ENDFUNCTION

```

Defines a record called OR within the function COPY\_ORD. Since the DEFINE statement is within a function definition, OR is a *local variable* and is only usable within the function COPY\_ORD.

## Example 8

```

DEFINE MYVAR : ORACLE VARCHAR2(20)
    CHARSET "ISO8859-1"

DEFINE CNAME : ODBC CHAR(10) ALLOW NULLS
    CHARSET "ANSI_X3.4-1968"

```

Defines a variable length string called MYVAR that will use the alternate character set ISO8859-1, and a fixed length string called CNAME that will use the alternate character set ANSI\_X3.4-1968 and

will allow `null` values.

**DELETE****Deletes the current record.**

The `DELETE` statement is used to delete the current record from a file that was read with a `READ` statement.

**Syntax**

```
DELETE read-tag
```

`read-tag` is an active read tag created with the `READ` statement. See the `READ` statement in this chapter for information on read tags. The `DELETE` statement deletes the current record read with the `READ` statement.

**Considerations**

The `DELETE` statement may only be used on files that support record deletion. See [Chapter Four, File Types](#), for more information.

When several files are involved, it is often necessary to make certain deletes are done in a particular order. For example, when deleting records from an `IMAGE` master dataset, all corresponding detail records must be deleted before the record can be deleted from the master dataset.

**Examples****Example 1**

```
OPEN ORDB IMAGE ORDERS
CREATE AR ARCHIVE ARCHFILE
READ ORD = ORDB.ORDERS FOR SHIP = "Y"
  COPY ORD TO AR.ORDERS
  READ LINES = ORDB.ORD-LINES &
    FOR ORD-NO = ORD.ORD-NO
  COPY LINES TO AR.ORD-LINES
  DELETE LINES
ENDREAD
DELETE ORD
ENDREAD
```

The above script reads, copies and deletes from the master dataset `ORDERS`, and the detail dataset `ORD-LINES`. Note that the delete for the records in the `ORDERS` file is after the delete for `ORD-LINES`.

**DIRECT****Direct Database Command**

The `DIRECT` statement is used to directly execute a database command. The `DIRECT` statement is database dependent and is used to execute SQL statements when accessing an SQL database.

**Syntax**

```
DIRECT db-tag, "statement"  
      [ ; IGNORE ERRORS ]  
      [ ; SHOW ROWCOUNT ]
```

`db-tag` is the tag of an SQL database that has been opened with the `OPEN` statement.

`statement` is an SQL statement that is executed immediately during script processing.

`IGNORE ERRORS` causes an error returned by `DIRECT` processing to be ignored. The `DIRECT` statement is executed as the script is processed. Normally, if any error occurs during script processing, then script execution after the `GO` statement is not done. `IGNORE ERRORS` causes an error message to be displayed, but allows script execution after `GO` to continue.

`SHOW ROWCOUNT` causes the number of rows affected by the SQL statement to be displayed.

**Considerations**

The `DIRECT` statement is executed as the script is processed. This allows the `DIRECT` statement to create tables or other objects that can be accessed later in the Warehouse script.

To use `DIRECT` as part of script execution after the `GO`, the `DIRECT` *function* should be called. See [Chapter Five, Warehouse Expressions](#) for details on the `DIRECT` function.

**Examples****Example 1**

```
OPEN DB ORACLE &  
      SCOTT/TIGER HOME=D:\ORANT SID=ORCL
```

```
DIRECT DB, &
```

```
"DROP TABLE CTEMP" IGNORE ERRORS
DIRECT DB, &
"CREATE TABLE CTEMP ( " &
    "CUSTNO      NUMBER(6) NOT NULL, " &
    "CUSTNAME    VARCHAR2(40), " &
    "TBALANCE    NUMBER(7,2), " &
    " ) "

DEFINE CREC : USING DB.CTEMP

READ C=DB.CUSTOMERS FOR PAYSTAT = "DEL"
    SETVAR CREC.CUSTNO      = CUSTNO
    SETVAR CREC.CUSTNAME    = CUSTNAME
    SETVAR CREC.TBALANCE    = 0
    READ U=DB.ORDERS &
        FOR CUSTNO = C.CUSTNO &
        AND BALANCE > 0
        SETVAR CREC.TBALANCE = &
            CREC.TBALANCE + BALANCE
    ENDREAD
    COPY CREC TO DB.CTEMP
ENDREAD

READ X=DB.CTEMP &
    FOR TBALANCE > 1000 &
    ORDER BY TBALANCE DESC
    PRINT CUSTNO, CUSTNAME, TBALANCE
ENDREAD

CALL DIRECT(DB, "DROP TABLE CTEMP")
```

The above script opens an Oracle database and uses a temporary table called CTEMP to add up and print customer balances. After the OPEN statement a DIRECT statement is used to drop the table CTEMP if it already exists. IGNORE ERRORS is specified because the drop will fail if the table does not already exist. Another DIRECT statement is used to create the table CTEMP. A DEFINE statement is used to create a Warehouse record of the same format as CTEMP. CUSTOMER and ORDERS records are then read to total the customer balance and write the information to the temporary table CTEMP. CTEMP is read ordered in descending order of balance for balances greater than \$1,000. After all CTEMP records have been read the DIRECT *function* is called to drop the table.

**END****Indicates the end of a construct.**

The `END` statement indicates the end of a multi-statement construct. The `END` statement can be used as an abbreviated form of `ENDIF`, `ENDFUNCTION`, `ENDREAD`, `ENDTRY`, or `ENDWHILE`.

**Syntax**

<code>END</code>	or
<code>ENDFUNCTION</code>	or
<code>ENDIF</code>	or
<code>ENDREAD</code>	or
<code>ENDTRY</code>	or
<code>ENDWHILE</code>	

**Considerations**

The `END` statement may be used as an abbreviation of `ENDFUNCTION`, `ENDIF`, `ENDREAD`, `ENDTRY` or `ENDWHILE` to indicate the end of a multi-statement construct.

**Examples****Example 1**

```
OPEN CUSTDB IMAGE CUSTDB.PUB &
  PASS=WRITER MODE=1
CREATE OUT ARCHIVE ARCFIL
READ CUST-MAST = CUSTDB.CUST-MAST &
  FOR STATUS = "CLOSED"
  COPY CUST-MAST TO ARCHIVE.CUST-MAST
  IF UPDATE-DATE > 940222 THEN
    PRINT CUST-NO, "retained."
  ELSE
    PRINT CUST-NO, "deleted."
    DELETE CUST-MAST
  END
END
```

The `END` statement is used in this example to terminate both the `IF` and `READ` statements.

**ESCAPE****Causes a user error condition.**

The `ESCAPE` statement causes a Warehouse error condition. If the error is not caught by a `RECOVER` statement, script execution terminates.

**Syntax**

```
ESCAPE [escape-message]
```

`escape-message` is an optional expression that describes the error condition. If the `ESCAPE` statement is caught by a `RECOVER` statement, the `escape-message` is accessible by accessing the Warehouse variable `$ERR.ESCMSG`. If there is no active `TRY/RECOVER` statement, then the `escape-message` is displayed on `stderr` before Warehouse terminates.

**Considerations**

The `ESCAPE` statement outside a `TRY` statement causes Warehouse to display an error message along with the `escape-message` then stop processing the script.

**Examples**Example 1

```
OPEN AR ORACLE SCOTT/TIGER
READ CUST = AR.CUSTOMERS
  TRY
    UPDATE CUST SET TOTAL = 0
  RECOVER
    ESCAPE "Error updating customer:" + &
      CUST_NUMBER
  ENDTRY
ENDREAD
```

This example opens an Oracle database and attempts to update each record from the `CUSTOMERS` table. If the `UPDATE` ever fails, the `ESCAPE` statement is executed, which causes an error message containing the customer number to be printed and then script processing is stopped since the `ESCAPE` statement is not caught by a `TRY`. If the `ESCAPE` statement were a `PRINT` statement instead, `CUSTOMER` records would continue to be read after printing the error message.



**EXIT****Exits Warehouse.**

The `EXIT` statement exits Warehouse and returns to the system prompt.

**Syntax**

`E[EXIT]`

**Considerations**

The `EXIT` statement immediately exits Warehouse back to the system. No further script processing is completed.

By using `EXIT` instead of `GO`, Warehouse checks the syntax and database access of the script, but does not execute the script.

**Examples****Example 1**

```
OPEN DB IMAGE SALES PASS=; MODE=1
READ CUSTS = DB.CUSTOMERS
    PRINT CUST-NO, CUST-NAME, ADDR
ENDREAD
EXIT
```

The `EXIT` statement causes Warehouse to immediately terminate without reading any records.

## FORMAT

**Creates a format definition for records and variables.**

The `FORMAT` statement creates a format definition to be used in a later `COPY`, `DEFINE`, `FORMAT`, or `READ` statement.

## Syntax

```
FORMAT format-name : item-type
```

where `item-type` is:

```
[db-type] simple-type
FORMAT format-name
RECORD item-list END
USING file
```

`format-name` is the name to be given to the format definition.

`db-type` specifies the database originator of the type. Examples of `db-type` are `ALLBASE`, `IMAGE`, and `ORACLE`. See [Chapter Six, Data Types](#), for details on specifying data types.

`simple-type` specifies the data type of the `format-name`. See [Chapter Six, Data Types](#), for details on specifying data types.

`FORMAT format-name` specifies that the format being created be exactly the same as a previously created format using the `FORMAT` statement.

`RECORD item-list` specifies the definition of a record. See [Chapter Six, Data Types](#), for details on specifying records.

`USING file` specifies that the format being defined has the same format as a file previously opened with the `OPEN` statement. This is typically in the format of `db-tag.table-name`.

## Considerations

The `FORMAT` statement differs from the `DEFINE` statement in that no storage is created with the `FORMAT` statement. A format is simply a data

description to be applied at a later time (e.g. with a `READ` or `DEFINE` statement); whereas, the `DEFINE` statement actually creates data storage that can be used immediately.

## Examples

### Example 1

```
FORMAT ORDER : RECORD
    ORDER-NUM : ALLBASE CHAR(16)
    CUST-NUM  : ALLBASE CHAR(16)
    DATE      : ALLBASE CHAR(8)
    ORD-STAT  : ALLBASE CHAR(8)
END
```

Describes a 48 byte record format called `ORDER` that has four `ALLBASE` type character elements: `ORDER-NUM`, `CUST-NUM`, `DATE`, `ORD-STAT`.

### Example 2

```
OPEN SEC IMAGE SECDB
FORMAT USER-FMT : USING SEC.USERS
```

Opens the `IMAGE` database `SECDB` using the database tag `SEC`. It then creates a format called `USER-FMT` that has the exact same format as the file `USERS` within the database `SEC`.

### Example 3

```
FORMAT COMPANY-REC2 : FORMAT COMPANY-REC
```

Creates a format called `COMPANY-REC2` that has the format of `COMPANY-REC` which was previously created by another `FORMAT` statement.

### Example 4

```
FORMAT LINE : ORACLE CHAR(80)
```

Creates a format called `LINE` that is an 80 byte Oracle type character string.

### Example 5

```
OPEN ORDDDB IMAGE ORDERS PASS=POWER
```

```
OPEN ORDFIL TEXT ORDERF
FORMAT ORDERFMT : RECORD
    ORDER-NUM : ALLBASE CHAR(16)
    CUST-NUM  : ALLBASE CHAR(16)
    DATE      : ALLBASE CHAR(8)
    ORD-STAT  : ALLBASE CHAR(8)
END
READ ORD = ORDFIL FORMAT ORDERFMT
    COPY ORD TO ORDDDB.ORDER-HEADER
ENDREAD
```

Opens the IMAGE database ORDERS and the text file ORDERF. A 48 byte record format ORDERFMT is defined that has four ALLBASE type character elements: ORDER-NUM, CUST-NUM, DATE, ORD-STAT. The text file ORDERF is read with the records being described by the format ORDERFMT. The records from ORDERF are then copied to the ORDER-HEADER dataset within the ORDERS database.

**FUNCTION****Defines a User-Defined Function.**

The `FUNCTION` statement begins the definition of a user-defined function. Function parameters are defined and all statements between `FUNCTION` and `ENDFUNCTION` are considered part of the function definition. If the function returns a value, it is called using a Warehouse expression. If the function does not return a value, it is called using the `CALL` statement.

**Syntax**

```
FUNCTION function-name  
    [ ( parameter-list ) ]  
    [ : return-type ]
```

Where parameter-list is:

```
[VAR] parm-name : parm-type  
[, [VAR] parm-name : parm-type ]  
[, ... ]
```

where parm-type and return-type are of the form:

```
[db-type] simple-type  
FORMAT format-name  
USING file
```

function-name is the name of the function being defined. The function-name may not be the same as db-tag or a `DEFINED` variable.

parm-name is the formal parameter name. If `VAR` precedes the parameter name, the parameter is passed to the function by *reference* and the actual parameter type must exactly match the parameter type specified in parm-type. If `VAR` is not specified, the actual parameter is passed by *value* after being converted to parm-type.

parm-type is the data type of the function parameter. It indicates the treatment of the parameter in the body of the function. See [Chapter Six, Data Types](#), for details on specifying data types.

`return-type` is optional and indicates the data type of the return value of the function. If `return-type` is specified, the function is called by using it in an expression, and there must be a `RETURN` statement in the body of the function that returns a value of the specified data type. If no `return-type` is specified, the function must be called with the `CALL` statement, and any `RETURN` statements cannot return a value. See [Chapter Six, Data Types](#), for details on specifying data types.

`db-type` specifies the database originator of the type. Examples of `db-type` are `ALLBASE`, `IMAGE`, and `ORACLE`. See [Chapter Six, Data Types](#), for details on specifying data types.

`simple-type` specifies the data type of the `format-name`. See [Chapter Six, Data Types](#), for details on specifying data types.

`FORMAT format-name` specifies a type that was created using the `FORMAT` statement.

`USING file` specifies a type that has the same format as a file previously opened with the `OPEN` statement. This is typically in the format of `db-tag.table-name`.

## **Considerations**

A `parameter-list` is optional in the function definition. If a function has a parameter list, all parameters must be present in the calling statement. Functions may not be defined that have a variable number of parameters.

Parameters may be passed either by value or by reference. To pass a parameter by reference the keyword `VAR` must precede the parameter name.

Parameters passed by `VAR` *must* have the exact same data type as specified in the function definition.

Parameters passed by value (the default) are converted, if necessary, to the data type specified in the function header. To pass a parameter by value, it need not have the same data type as in the function definition, but it must be of the same data type family (e.g. numeric, string).

The following statements are not permitted within a FUNCTION definition: OPEN, CREATE, FORMAT, FUNCTION.

A function may not be defined within an IF statement, READ loop, WHILE loop, or another function definition.

The DEFINE statement may be used to create local variables within the function. The DEFINE statement is only supported when it is the first statement in the function or preceded only by other DEFINE statements.

If a predefined Warehouse function and a user-defined function have the same name (e.g. CONVERT), the user-defined function overrides the Warehouse function and the Warehouse function becomes inaccessible.

Function recursion is supported, but READ statements within a recursive function may or may not work, depending on the type of database read.

## Examples

### Example 1

```
OPEN ODB IMAGE ORDS PASS=READR MODE=5

FUNCTION NUMORDS (CN : IMAGE X8) &
    : INTEGER
    * Counts the number of orders for CN

    DEFINE NUM : INTEGER
    SETVAR NUM = 0
    READ C = ODB.ORDERS FOR CUSTNO = CN
        SETVAR NUM = NUM + 1
    ENDREAD
    RETURN NUM
ENDFUNCTION
```

```
READ C = ODB.CUSTOMERS
  PRINT CUSTNO,CUSTNAME,NUMORDS(CUSTNO)
ENDREAD
```

This example opens the IMAGE database ORDS using a tag of ODB, a password of READR and an open mode of 5, permitting read-only access. A function is defined called NUMORDS that has a formal parameter of type IMAGE X8 called CN. NUMORDS returns an integer to the caller. A DEFINE statement is used to define a local integer variable called NUM that is used to count the number of orders in the ORDERS dataset. Records are then read from ORDERS for the matching customer number. NUM is incremented for each record read. After the number of orders has been calculated NUM is returned to the caller.

Outside of the function definition, the CUSTOMERS dataset is read and the customer number, customer name and number of orders placed by the customer are printed.

### Example 2

```
OPEN ODB IMAGE ORDERS PASS=RDR MODE=5

FUNCTION FIND_BIGGEST &
  (CN : IMAGE X10, &
   VAR BIGGEST : IMAGE R4)
  * Finds largest order for customer CN

  SETVAR BIGGEST = 0

  READ OL = ODB.ORDLINES FOR CUST = CN
    IF PRICE * QTY > BIGGEST
      SETVAR BIGGEST = PRICE * QTY
    ENDIF
  ENDREAD
ENDFUNCTION

DEFINE BIGGEST : IMAGE R4
READ CUST = ODB.CUSTOMERS
  CALL FIND_BIGGEST(CUSTNO, BIGGEST)
  IF BIGGEST > 10000
    PRINT CUSTNO, CUSTNAME, BIGGEST
```



```
ENDIF  
ENDREAD
```

This example opens the IMAGE database ORDERS using a password of RDR and an open mode of 5. A function called FIND\_BIGGEST is defined that reads all ORDLINES for a given customer number and calculates the largest order placed by that customer. FIND\_BIGGEST takes two parameters, the customer number CN by value, and BIGGEST which is passed by reference. The CUSTOMERS dataset is read and if the biggest order the customer has placed is over \$10,000, the CUSTNO, CUSTNAME, and size of the biggest order are printed.

### Example 3

```
FUNCTION GETDESC(PN : IMAGE X12) &  
    : IMAGE X48  
  
    READ P = PDB.PARTMASTER &  
        FOR PARTNO = PN  
        RETURN PARTDESC  
    ENDREAD  
    RETURN "UNABLE TO FIND PART " || PN  
ENDFUNCTION
```

The above function takes a part number as an IMAGE X12 parameter and looks up the part description in the database. If the part is found, the description is returned to the caller. If the part number is not found, an error message is returned instead.

### Example 4

```
FUNCTION UPSCUST &  
    (CUSTREC : USING CDB.CUSTOMERS) &  
    : USING CDB.CUSTOMERS  
  
    DEFINE CR : USING CDB.CUSTOMERS  
  
    SETVAR CR = CUSTREC  
    SETVAR CR.CUSTNAME = UPS(CR.CUSTNAME)  
    SETVAR CR.CUSTADDR = UPS(CR.CUSTADDR)
```

```
        SETVAR CR.CUSTCITY = UPS(CR.CUSTCITY)

        RETURN CR
    ENDFUNCTION
```

The UPSCUST function defined above takes a customer record called CUSTREC that has the exact same layout as the CUSTOMERS table from the database opened as CDB and returns a record with the same format. Another record CR is defined (also with the CUSTOMERS format) as a local variable within UPSCUST and CR is set to be equal to CUSTREC. The customer name, address and city fields are then upshifted in the CR record using the UPS function. The CR record is returned to the caller.

#### Example 5

```
FUNCTION UPSCUST &
    (VAR CR : USING CDB.CUSTOMERS)

    SETVAR CR.CUSTNAME = UPS(CR.CUSTNAME)
    SETVAR CR.CUSTADDR = UPS(CR.CUSTADDR)
    SETVAR CR.CUSTCITY = UPS(CR.CUSTCITY)
ENDFUNCTION
```

This UPSCUST function has basically the same functionality as the previous function, except that a VAR parameter is used. This allows the customer name, address and city to be upshifted within the record itself, rather than returning a record with the upshifted values.

<b>GO</b>	<b>Begins processing of the script.</b>  The GO statement terminates the script definition phase and begins processing the script. At script completion, Warehouse terminates.
<b>Syntax</b>	GO
<b>Considerations</b>	<p>A script can only contain one GO statement.</p> <p>At the conclusion of the script, statistics are printed including information on the execution time and number of records read, copied, updated and deleted. After execution of the script, all files are closed and Warehouse terminates.</p> <p>If there were any errors during script processing, no script execution takes place and Warehouse terminates with the system error flag set. (On MPE, the job control word JCW is set to FATAL.)</p> <p>If no errors are detected during script processing, the script is executed. If there are errors during script execution, Warehouse terminates with the system error flag set. If no errors occur, Warehouse terminates normally at the end of script execution.</p> <p>A GO statement is <i>not</i> required when a script file name is included as a parameter when Warehouse is run. In that case, Warehouse automatically executes the script when the end of the script file is reached.</p>
<b>Examples</b>	None.

## HEADER

**Generates a page header line.**

The `HEADER` statement is used to create page header lines that are displayed at the top of each print page.

## Syntax

```
HEADER [print-list] [comma-semi]
```

or

```
HEADER [rpt-tag][print-list][comma-semi]
```

where `print-list` is:

```
print-item [comma-semi print-item...]
```

where `print-item` is one of the following:

<code>\$CENTER</code>	or
<code>expression [fmt]</code>	or
<code>item [fmt]</code>	or
<code>\$NEW</code>	or
<code>\$PAGENO [fmt]</code>	or
<code>\$TAB column</code>	

where `fmt` is either:

<code>: width</code>	or
<code>PIC "picture"</code>	

`comma-semi` is either a comma (,) or a semicolon (;) used as a separator. When a comma is used to separate print items, a space is printed between items. When a semicolon is used to separate print items, no space is printed between items. A `HEADER` statement with a trailing comma or semicolon causes the header line to be continued, with items from the next `HEADER` statement going on the same line.

`rpt-tag` is the name of the file tag used in the `OPEN` statement to open the report file. Note that the `rpt-tag` must be enclosed in square brackets ([ ]). When `rpt-tag` is specified, the `HEADER`

statement applies only to the indicated report file.

`$CENTER` indicates that the following header items are to be centered on the print page. A `$TAB` item causes items to no longer be centered. The page width may be modified with the `SET` statement. Only one `$CENTER` is permitted per `HEADER` statement.

`expression` is any string or variable expression composed of variables and constants. See [Chapter Five, Expressions](#), for more information.

`$NEW` indicates that a new page header is desired and the old page header is to be discarded. `$NEW` is used when two or more report sections are desired.

`$PAGENO` indicates that the current page number is to be displayed.

`$TAB column` indicates that the next header item is to be printed in the column specified by `column` with column 1 being the leftmost column.

`width` and `picture` are used to specify item formatting. See the `PRINT` statement for more information.

## Considerations

Multiple `HEADER` statements can be used to create multiple lines of page header.

The page header is printed at the top of every page. The page length can be modified using the `SET` statement.

## Examples

### Example 1

```
HEADER $CENTER, "Taurus Software",  
HEADER $TAB 70, "Page", $PAGENO : 4  
HEADER $CENTER, "Customer Report"  
HEADER
```

This example creates a three line header. The first line has `Taurus Software` centered on the page with `Page` and the page number starting in column 70. The page number is four print positions wide. (The second `HEADER` statement is a continued on the same print line as the first `HEADER` statement because the first statement ends in a comma.) The second line has `Customer Report` centered on the page, and the third header line is blank.

**HELP****Displays help information.**

The `HELP` statement is used to display help on Warehouse statements or other topics.

**Syntax**

`HELP [topic]`

`topic` is either a Warehouse statement or a help topic. If no `topic` is entered, a list of available topics is displayed.

**Examples****Example 1**

```
1> HELP DELETE
DELETE read-tag

    Deletes a record from a database.
    Requires read-tag from a previous READ statement.

2>
```

This example displays help on the `DELETE` statement.

**IF**

**Defines the beginning of an IF block.**

The IF statement defines the beginning of a conditional IF block. The IF statement may have an optional ELSE block.

**Syntax**

```
IF condition [THEN]
    statement-list
[ELSE [IF condition [THEN]]]
    statement-list
ENDIF
```

`condition` specifies a Boolean expression to be evaluated. It may be any Boolean expression containing arithmetic and string operators. If the expression evaluates to TRUE, the `statement-list` following the IF statement is executed, otherwise the `statement-list` following the ELSE statement is executed (if there is an ELSE). If the condition is FALSE and there is no ELSE block, control skips to the ENDIF statement. For more information on Boolean expressions, see [Chapter Five, Expressions](#).

`statement-list` is a list of Warehouse statements that may include IF statements.

**Considerations**

The ELSE statement may contain an optional IF condition that is executed when control falls to the ELSE and the condition on the ELSE is TRUE.

**Examples****Example 1**

```
OPEN AR ARCHIVE ARCHFILE
DEFINE STATUS : ORACLE CHAR(16)
READ CUST = AR.CUSTOMERS
  IF CUST-STAT = "HD"
    SETVAR STATUS = "HOLDING"
  ELSE IF CUST-STAT = "IP"
    SETVAR STATUS = "IN PROCESS"
  ELSE IF CUST-STAT = "CL"
    SETVAR STATUS = "CLOSED"
  ELSE
    SETVAR STATUS = "UNKNOWN"
  ENDIF
PRINT CUST-NO, CUST-NAME, STATUS
```



ENDREAD

This example shows how the `IF` statement combined with `ELSE` `IF` may be used to select from several expressions.

**LIST****Lists information.**

The `LIST` statement displays a list of databases, files, variables, and formats. `LIST` is also used to display a list of tables within a database or archive file.

**Syntax**

`LIST`

or

`LIST [db-tag]`

`db-tag` specifies database tag or archive file tag for which the tables are to be listed. All tables (files, or datasets) which belong to the database specified by `db-tag` are listed.

If no `db-tag` is specified, all databases, files, variables and formats are listed.

**Considerations**

The `LIST` statement is executed immediately and is not executed as part of running the script.

The `LIST` statement is used to display a general list of items. The `SHOW` statement may be used to obtain specific information about any item displayed by the `LIST` statement.

**Examples****Example 1**

```
OPEN ODB ORACLE SCOTT/TIGER
CREATE TMP TEXT MYTMP
DEFINE SUBTOT : ORACLE NUMBER
FORMAT TMPFMT : RECORD
      ORNO : ORACLE CHAR(12)
      CNO  : ORACLE CHAR(12)
END
LIST
```

Variable definitions:

```
      SUBTOT      : ORACLE NUMBER
```

Files and Databases:

```
      ODB         : ORACLE DB
```

```
TMP          : TEXT FILE
```

```
FORMAT Statements:
```

```
TMPFMT      : RECORD
```

This example shows an Oracle database opened, a text file called MYTMP created, a variable SUBTOT defined, and a format called TMPFMT created. The LIST statement is used to display a list of items known to Warehouse.

### Example 2

```
OPEN TDB IMAGE TESTDB PASS=MYPASS MODE=3
LIST TDB
```

Dataset Name	Type	Entries	Capacity
COMPANY	D	6415	8000
USERS	M	568	1009
FORECAST	D	2781	6000
.			
.			
.			

This example opens the IMAGE database TESTDB using a database tag of TDB, a password of MYPASS and an open mode of 3, for exclusive access. The LIST statement is used to display all datasets within TESTDB.

### Example 3

```
1> OPEN ORAIN ORACLE scott/tiger
2> LIST ORAIN
```

TABLE NAME	RECORDS	PAGES
-----	-----	-----
SYS.DUAL		
SYS.SYSTEM_PRIVILEGE_MAP		
SYS.TABLE_PRIVILEGE_MAP		
.		
.		
.		
SCOTT.DEPT		
SCOTT.EMP		
SCOTT.BONUS		
SCOTT.SALGRADE		
.		
.		
.		

This example defines an Oracle data source to

open, then lists all its tables.

#### Example 4

```
1> OPEN SQLIN ODBC MyPTA
2> LIST SQLIN
```

TABLE NAME	OWNER	TYPE
-----	-----	----
CALLIST	dbo	TABLE

This example opens a SQL Server data source through an ODBC connection and lists its tables.

#### Example 5

```
1> OPEN db REMOTE CENTRIC &
2> USER=XXXX PASS=XXXX &
3> DB2 db2matest DB2INSTANCE=db2 &
4> DB2DIR="C:\Program Files\IBM\SQLLIB"
5> LIST db
```

TABLE NAME	OWNER	TYPE
-----	-----	----
ANCHOR	ADMINISTRATOR	TABLE
ATABA	ADMINISTRATOR	TABLE
ATABATEST	ADMINISTRATOR	TABLE
. . .		
CUSTOMER	EAPL1204	TABLE
CUSTOMER_ADDR	EAPL1204	TABLE
. . .		

This example opens a DB2 data source called db2matest on a remote machine called CENTRIC and lists all structures found there.

**OPEN****Opens an existing database or file**

The `OPEN` statement opens a database or file to be accessed within the Warehouse script.

**Syntax**

```
OPEN db-tag file-type [file-parms]
```

`db-tag` is the database tag used to reference the database or file in the remainder of the script.

`file-type` is the type of database or file. The file types supported on your version vary according to operating system and purchase options. Supported values of `file-type` are:

<code>ALLBASE</code>	Allbase DBE file
<code>ARCHIVE</code>	Warehouse archive file
<code>CSV</code>	Comma separated file
<code>DB2</code>	IBM DB2 database
<code>FIXED</code>	Fixed record length file
<code>IMAGE</code>	IMAGE database
<code>ODBC</code>	ODBC SQL Server database
<code>ORACLE</code>	Oracle database
<code>REMOTE</code>	Remote access file
<code>REPORT</code>	Report file
<code>TEXT</code>	Text (character) file

See [Chapter Four, File Types](#), for more information.

`file-parms` is the file name of the database or file to be opened and any parameters specific to `file-type` that are needed to open the database or file. The exact meaning of `file-parms` depends on the `file-type`. See [Chapter Four, File Types](#), for more information.

**Considerations**

The `OPEN` and `CREATE` statements perform the same basic function, i.e., make a file or database available for access by Warehouse. The difference is that `OPEN` opens an existing database or file, and `CREATE` makes a new one.

CREATE and OPEN statements are generally placed at the top of your script file because files must be opened before they may be referenced within the script.

## **Examples**

### Example 1

```
OPEN ODB IMAGE ORDERS.PUB.DB &  
    PASS=MYPASS MODE=1
```

This example opens the IMAGE database ORDERS.PUB.DB using a password of MYPASS and an open mode of 1, allowing read-write access. The database tag ODB is used to access this database in the remainder of the script.

### Example 2

```
OPEN A ARCHIVE ARCHFILE
```

This example opens ARCHFILE as an input file using a database tag of A to reference the archive file.

### Example 3

```
CREATE RT REMOTE PEAR.TAURUS.COM &  
    USER=mary PASS=mpw &  
    TEXT D:\udata\mary\wh\scriptlog
```

This example creates a text file scriptlog in the directory D:\udata\mary\wh\ on the remote Windows server PEAR.TAURUS.COM. The remote user is logged on as user mary using the password mpw.

### Example 4

```
OPEN MYREP REPORT REPFIL  
HEADER [MYREP] $CENTER, &  
    "ACCOUNT SUMMARY"
```

This example opens the report file REPFIL and creates a page header for it using the HEADER statement.

### Example 5

```
OPEN db REMOTE CENTRIC &  
  USER=myuser PASS=mypass &  
  DB2 db2matest DB2INSTANCE=db2 &  
  DB2DIR="C:\Program Files\IBM\SQLLIB"
```

This example opens a DB2 data source called db2matest on a remote machine called CENTRIC.

### Example 6

```
OPEN db ORACLE scott/tiger@zoo
```

This example opens a local ORACLE connection using tns (transparent network substrate) redirection to a remote ORACLE data source. The local ORACLE instance must have an entry in `tnsnames.ora` called zoo that describes the connection redirection to the remote box. The user/password provided is used to connect to the remote zoo database.

**PRINT****Generates a print line.**

The PRINT statement is used to display record elements and variables to standard output or to a report file.

**Syntax**

```
PRINT [print-list] [comma-semi]
```

or

```
PRINT [[rpt-tag]][print-list] [comma-semi]
```

where print-list is:

```
print-item [comma-semi print-item...]
```

where print-item is one of the following:

```
expression [fmt]           or  
item [fmt]                  or  
$PAGE                       or  
$TAB column
```

where fmt is either:

```
: width                      or  
PIC "picture"
```

comma-semi is either a comma (,) or a semicolon (;) used as a separator. When a comma is used to separate print items, a space is printed between items. When a semicolon is used to separate print items, no space is printed between items. A PRINT statement with a trailing comma or semicolon causes the print line to be continued, with items from the next PRINT statement going on the same line.

[rpt-tag] is the name of the file tag used in the OPEN statement to open the report file. Note that the rpt-tag must be enclosed in square brackets ([ ]). When [rpt-tag] is specified, the PRINT statement applies only to the indicated report file.



`expression` is any string or variable expression composed of record elements, variables, or numeric or string constants. See Chapter Five, Expressions, for more information.

`$PAGE` Indicates that a page break is to be inserted into the report. Subsequent `PRINT` output is printed starting at the top of the next print page.

`$TAB column` indicates that the next print item is to be printed in the column specified by `column` with column 1 being the leftmost column.

`width` specifies the width of the print field. Numeric type items are right justified and all other items are left justified.

`picture` is a string of characters that indicate the output format of the item being printed. The interpretation of the picture depends on the data type of the item being printed. For numeric items the picture is similar to the COBOL `PICTURE` clause. For date type items, the picture is similar to the date formatting in Oracle.

#### Numeric Pictures

When formatting a number item, each character of the picture represents a character position in the output. A character may be replicated by entering a number in parentheses after the character, e.g. `x(6)` is the same as `xxxxxx`. The picture characters are as follows:

##### Numeric `PIC` characters

- `z` If a `z` matches a leading zero in the field's content, it is replaced by a blank. If not, `z` is replaced by a digit in the field's content.
- `9` Each `9` is replaced by one digit from the field's content.
- `*` If an asterisk matches a leading zero in the field's content, a `*` is placed in that character position. If not, it is replaced by

a digit from the field's content.

- 0 A 0 is placed in that character position.
- ,
- If all digits to the left of the comma are suppressed zeros, the comma is replaced by a blank. If not, a comma is inserted in that character position.
- .
- A decimal point is inserted in that character position. An edit string for a numeric field can contain only one period.
- 
- If only one minus sign (-) is specified, it is replaced by either a blank (if the field's content is positive) or a minus sign (if it is negative). If more than one minus sign is specified, then the minus sign is floating.
- +
- If only one plus sign (+) is specified, it is replaced by either a plus sign (if the field's content is positive) or a minus sign (if it is negative). If more than one plus sign is specified, then the plus sign is a floating sign.
- \$
- If only one dollar sign is specified, it is replaced by a \$ in that character position. If more than one dollar sign is specified, the dollar sign is floating.
- CR
- If the field's content is positive, the letters CR are inserted. If the field's content is negative, CR is replaced by 2 blanks.
- DB
- If the field's content is negative, the letters DB are inserted. If the field's content is positive, DB is replaced by 2 blanks.
- %
- A percent sign is inserted in that character position.

#### Date Pictures

When formatting a date item, each token represents one element of a date or time. A list of

the picture tokens can be found in [Chapter 5: Warehouse Expressions - Built-In Functions](#), **DATE2STR**.

## Examples

### Example 1

```
PRINT "CUST-NO", $TAB 20, "CUSTOMER  
NAME"  
PRINT CUST-NO, $TAB 20, CUST-NAME
```

Prints CUST-NO at column 1, and CUSTOMER-NAME at column 20 as column headings, then prints the CUST-NO field followed by CUST-NAME field.

### Example 2

```
PRINT PART-NO:8; " ":6; PART-DESC:30
```

Prints the PART-NO field using a width of 8 characters, followed by 6 spaces, followed by PART-DESC using 30 characters. Note the use of the semicolon separator to suppress the space between items. If commas had been used instead of semicolons, the part number would have been separated from the description by 8 spaces instead of 6.

### Example 3

```
OPEN MYRPT REPORT RPTFILE  
PRINT [MYRPT] CUST-NO,CUST-NAME : 30,  
PRINT [MYRPT] BALANCE PIC "Z(7)9.99"
```

Prints CUST-NO, then CUST-NAME using a width of thirty characters, followed by BALANCE all on the same line. BALANCE is formatted with up to seven leading zeros displayed as blanks, followed by a digit, followed by a decimal point and two digits after the decimal point. Both lines are printed to the report file RPTFILE using the tag MYRPT.

### Example 4

```
PRINT 1, 1234.567 PIC "9(8)"
```

```
PRINT 2, 1234.567 PIC "9(8).99"  
PRINT 3, 1234.567 PIC "Z(7)9.99"  
PRINT 4, 1234.567 PIC "+Z(7)9.99"  
PRINT 5, 1234.567 PIC "+(7)9.99"  
PRINT 6, 1234.567 PIC "$ (7)9.99"  
PRINT 7, 1234.567 PIC "$ (7)9.99+"  
PRINT 8, -1234.567 PIC "**(7)9.99+"  
PRINT 9, 1234.567 PIC "-Z,ZZZ,ZZZ"  
PRINT 10, -1234.567 PIC "--Z,ZZZ,ZZZ.99"
```

```
1 00001235  
2 00001234.57  
3      1234.57  
4 +      1234.57  
5      +1234.57  
6      $1234.57  
7      $1234.57+  
8 ****1234.57-  
9      1,235  
10     -1,234.57
```

This example shows a series of PRINT statements with PICS and the resulting output.

#### Example 5

```
PRINT $NOW PIC "DY, Mon DD, YYYY"
```

Prints the current time as calculated by the \$NOW system variable, e.g. "Fri, Aug 30, 1996"

#### Example 6

```
PRINT $NOW PIC "HH24:MI:SS TZH:TZM"
```

Displays "11:28:12 -08:00"

**READ****Reads records from a database or file.**

The READ statement begins a loop that reads records from a database or file. The READ loop must be terminated by an ENDREAD statement.

**Syntax**

```
READ read-tag = file
    [FORMAT format-name]
    [FOR condition]
    [ORDER BY order-list]
```

```
    statement-list
```

```
ENDREAD
```

where order-list is:

```
    order-item [ASC | DESC] [, ... ]
```

`read-tag` is the name of the read tag created by the READ statement. The read tag name is used to access the data record within the read loop and to perform operations such as update and delete. The read tag is also used to access fields within the record read by the READ statement using the syntax `read-tag.field-name`.

`file` is the identifier of the file from which the data is to be read. This is typically in the format of `db-tag.table-name`. See [Chapter Four, File Types](#), for more information.

`format-name` is the name of a format previously created with the FORMAT statement. When `format-name` is specified, the format of the records read from the `file` is redefined to be of the format `format-name`.

`condition` specifies the condition upon which processing of the READ loop continues. If the `condition` is TRUE, then the statements within the READ loop are executed. If the `condition` is FALSE then the next record in `file` is read and tested for `condition`. If no `condition` is specified, all records in `file` are read serially. The

`condition` may be any valid Boolean expression. For more information on Boolean expressions, see [Chapter Five, Expressions](#).

`order-item` is the name of a field within the record being read. When an `ORDER BY` clause is specified, records are read in the specified sort order. If `ASC` (the default) is specified, the records selected by the `FOR` condition are read in ascending order. If `DESC` is specified, the records are read in descending order.

`statement-list` is a list of Warehouse statements that are executed for every qualifying record read. `statement-list` may include other `READ` statements.

### Considerations

When no `condition` is specified, all records in the `file` are read serially and processed by the statements within the `READ` loop.

When an end of file condition is encountered by the `READ` statement, control is passed to the next statement after the corresponding `ENDREAD` statement.

Field names inside the `FOR` condition or the body of a `READ` loop are accessed using the syntax: `read-tag.field-name`. Field names may be abbreviated to simply `field-name` in the `FOR` condition or the body of the *active* `READ` statement.

Warehouse attempts to optimize `condition` to achieve maximum performance. For example, when reading an `IMAGE` dataset Warehouse performs a keyed read whenever possible.

When the `condition` or part of the `condition` contains variables or fields *not* in the record being read, Warehouse may terminate the read loop when the condition is `FALSE`. This is known as a "shortened read" and is done to optimize performance. For example, the statement

```
READ R = DB.FILE FOR COUNTER <= 100
```

where COUNTER is a Warehouse variable, terminates as soon as an end of file is reached or COUNTER is greater than 100. Once COUNTER is greater than 100, Warehouse recognizes that reading more records will not make the condition true, so the read loop is terminated.

## **Examples**

### Example 1

```
OPEN ORDDB IMAGE ORDER.DB PASS=; MODE=1
CREATE TAPE ARCHIVE ORDTAPE
READ ORDERS = ORDDB.ORDERS &
  FOR CMPL = "Y" AND ORDER-NO > 1000
```

This example reads the ORDERS dataset in the ORDDB database looking for matches on the items CMPL and ORDER-NO. Any records which have a value of Y in the CMPL field and an ORDER-NO greater than 1000 are processed by statements following the READ statement and before the ENDREAD statement.

### Example 2

```
OPEN CUST ARCHIVE CUSTFILE
OPEN TEST ALLBASE TESTDBE
READ C = CUST.CUSTOMER
  COPY C TO TEST.TESTDB.CUSTS
ENDREAD
```

This example reads all records in the CUSTOMER file from the archive file CUSTFILE and copies them to the TESTDB.CUSTS table in the ALLBASE database TESTDBE.

### Example 3

```
OPEN CUSTDB IMAGE CUSTDB.DATABASE
READ CUST = CUSTDB.CUST-M &
  FOR STR(CUSTOMER,1,6) = "123456"
```

This example reads the CUST-M dataset in the

CUSTDB database and selects every record where the first six characters of the CUSTOMER data item are 123456. Notice that all functions and operations are supported by the READ statement. For more information on functions, see [Chapter Five, Expressions](#).

#### Example 4

```
OPEN PROD IMAGE PROddb &  
    PASS=READER MODE=5  
OPEN TEST IMAGE TESTDB &  
    PASS=WRITER MODE=3  
DEFINE NUM : I2  
SETVAR NUM = 1  
READ M = PROD.MASTER-SET FOR NUM <= 100  
    COPY M TO TEST.MASTER-SET  
    SETVAR NUM = NUM + 1  
ENDREAD
```

This example opens the IMAGE database PROddb using a password of READER and an open mode of 5 for read-only access and opens the database TESTDB using a password of WRITER and an open mode of 3 for exclusive access. A record counter NUM is defined to count the records as they are read. The dataset MASTER-SET from the database PROD is read serially as long as counter NUM is less than or equal to 100. This limits the selection to 100 MASTER-SET records. The selected MASTER-SET records are then copied to the test database and the counter variable NUM is incremented by 1.

#### Example 5

This example is a little more complex and demonstrates nested READ statements with conditions, key matches, and ordering.

The input database has the following structure:

Table: OFFER\_Q

Columns: CHANGE\_ID, STATUS\_NOW,  
TABLE\_NAME

Table: OFFERS\_CL



Columns: CHANGE\_ID, OFFER\_NO, TYPE

The output database has the following structure:

Table: TBL\_2\_UPD

Columns: CHG\_LOG\_ID, ACTION\_CD,  
OFFER\_NO, TYPE\_, etc.

We would like to read the input database OFFER\_Q looking for transactions that match a specific condition. When found, we want to read their corresponding entries on the OFFERS\_CL table. Then using information on the OFFERS\_CL table, make updates to the output database's TBL\_2\_UPD matching column names.

```
Step 1  OPEN InDB REMOTE BOX1 USER=## PASS=## &
        ODBC MyODBC1 USER=## PASS=##

Step 2  OPEN OutDB BOX2 USER=## PASS=## &
        ODBC MyODBC2 USER=## PASS=##

Step 3  READ InDB_QR = InDB.{dbo}.OFFER_Q &
        FOR STATUS_NOW = "NEW" &
        ORDER BY CHANGE_ID

Step 4    IF TABLE_NAME = "dbo.OFFERS"

Step 5      READ InDB_CLR = InDB.{dbo}.OFFERS_CL
            FOR CHANGE_ID = InDB_QR.CHANGE_ID

Step 6      IF InDB_QR.CHANGE_TYPE = "U"

Step 7      READ OutDB_R = &
            OutDB.{dbo}.TBL_2_UPD &
            FOR SLOTID = InDB_CLR.SLOTID

Step 8      UPDATE OutDB_R SET &
            CHG_LOG_ID = 0, &
            ACTION_CD  = "UPD", &
            OFFER_NO   =
                InDB_CLR.U_OFFERNO, &
            TYPE_      =
                InDB_CLR.U_TYPE, &
            (more assignments go here)

Step 9      ENDREAD
            ELSE IF InDB_QR.CHANGE_TYPE = "#"
            . . .
```

```
        ENDIF  
        ENDREAD  
    ELSE IF TABLE_NAME = "####"  
        . . .  
    ENDIF  
ENDREAD
```

Step1: We have an ODBC connection called MyODBC1 that points to a remote database on BOX1. We are going to refer to this database in the script as InDB.

Step 2: We have an ODBC connection called MyODBC2 that points to a local database on BOX2. We are going to refer to this database in the script as OutDB. Because this is a local connection, we can assume that this script is running locally on BOX2.

Step 3: Start a READ loop called InDB\_QR that points to a table on InDB called OFFER\_Q. We only want to pull rows from that table that have NEW in the column STATUS\_NOW. The matching records will be sorted by CHANGE\_ID.

Step 4: Using the above set of returned records, we want to process only records that have dbo.OFFERS in the column TABLE\_NAME through the next block of code.

Step 5: Start another READ loop called InDB\_CLR that points to a table on InDB called OFFERS\_CL. We only want to pull rows from that table where the CHANGE\_ID entry matches the CHANGE\_ID on the InDB\_QR READ loop.

Step 6: If the CHANGE\_TYPE column on the InDB\_QR contains a U then process updates to the output table.

Step 7: Start another READ loop called OutDB\_R that points to a table on OutDB called TBL\_2\_UPD. We only want to pull rows for updating from the output table where the SLOTID entry matches the

SLOTID on the InDB\_CLR READ loop.

Step 8: Update the records on the OutDB\_R READ loop by changing their column entries using data from the InDB\_QR READ entries.

Step 9: Terminate the READ loops, and the IF statements. Other tests would be made here for additional CHANGE\_TYPE entries like inserts, or deletes, and for additional TABLE\_NAME values.

**RETURN****Returns from a User-Defined Function.**

The RETURN statement exits from a user-defined function, and may return a value to the calling expression.

**Syntax**

```
RETURN [ return-value ]
```

return-value is an expression representing the value returned to the calling expression.

**Considerations**

If the function definition specifies a return-type, the RETURN statement is required to return a value and it is an error to reach the ENDFUNCTION statement without returning a value.

If the function definition does not specify a return-type, the ENDFUNCTION statement may be used to exit the function. Without a return-type in the function definition, the RETURN statement may not return a value.

**Examples**Example 1

```
FUNCTION CHKDATE(DT : X6) : X6
  IF DT = "      " THEN
    RETURN "890101"
  ELSE
    RETURN DT
  ENDIF
ENDFUNCTION
```

The function CHKDATE takes an IMAGE X6 item as a parameter called DT and returns an IMAGE X6 item. If DT is all spaces, 890101 is returned, otherwise DT is returned.

Example 2

```
FUNCTION CHKDATE(VAR DT : X6)
  IF DT <> "      " THEN
    RETURN
  ENDIF
  SETVAR DT = "890101"
ENDFUNCTION
```

The function `CHKDATE` takes an `IMAGE X6` item as a parameter called `DT`. The `VAR` keyword indicates `DT` is passed by reference. If `DT` is not all spaces, the function immediately returns. Otherwise `DT` is set to 890101 and the function returns automatically at the `ENDFUNCTION` statement.

**ROLLBACK****Causes a transaction Rollback.**

The ROLLBACK statement causes a Warehouse rollback operation. The ROLLBACK statement causes all databases supporting rollback that have been accessed during the transaction to be rolled back to the state when the transaction started.

**Syntax**

ROLLBACK

**Considerations**

In a typical script the ROLLBACK statement is unnecessary. The ROLLBACK statement is only provided for sophisticated transaction management and may cause problems if used inappropriately.

The ROLLBACK statement performs a rollback operation on all databases accessed by the script since the previous commit operation. There is no way to rollback a single database.

The effect of a ROLLBACK statement depends on the type of database accessed. The effect is as follows for each database type:

ALLBASE	Performs database rollback
ARCHIVE	No effect
CSV	No effect
DB2	Performs database rollback
FIXED	No effect, unless MPE/iX message file opened with NDR, in which case most recent record read is preserved for next read.
IMAGE	Calls DBXUNDO if locking is ROLLBACK.
ODBC	Performs database rollback
ORACLE	Performs database rollback
REMOTE	Depends on underlying database
REPORT	No effect
TEXT	No effect

**Examples**Example 1

```
OPEN RI &  
  REMOTE MPESYS USER=MGR.DBMGR &
```

```
IMAGE IMGDB PASS=IMPASS MODE=5
OPEN LO ORACLE SCOTT/TIGER

READ CUST = RI.CUSTOMERS
TRY
  COPY CUST TO LO.CUSTOMERS
  READ CT = RI.CUST-TRANS &
    CUSTNO = CUST.CUSTNO
  COPY CT TO LO.CUST_TRANS
ENDREAD
RECOVER
  * Rollback CUSTOMERS record and
  * any CUST_TRANS records
ROLLBACK
ENDTRY
ENDREAD
```

This example opens a remote Image database IMGDB on the MPE/iX system MPESYS. It then opens a local Oracle database. The CUSTOMERS dataset is read from the remote Image database. A TRY statement is used to catch any errors encountered while copying records to the local Oracle database. The CUSTOMER record is copied and matching CUST-TRANS records are read from the Image database and copied to the Oracle database. If an error occurs while copying, the RECOVER block is executed which causes the CUSTOMER and CUST\_TRANS records to be rolled back out of the local Oracle database. Also see [Chapter 4 Image Files Types: Set Locking](#).

**SET****Sets system and database options.**

The `SET` statement establishes values for system and database options.

**Syntax**

```
SET global-option value
```

or

```
SET db-tag db-option value
```

`global-option` specifies the name of the global option to be changed. `global-option` must be one of:

<code>AUTOPAD</code>	Automatically pads fixed length strings
<code>COMMITRATE</code>	Sets frequency of commits
<code>MSGs</code>	Sets display of conversion messages
<code>PAGELLENGTH</code>	Sets report page length
<code>PAGEWIDTH</code>	Sets report page width
<code>PRINTNULL</code>	Changes value displayed when a null value is printed
<code>PROGRESS</code>	Sets display of progress messages
<code>START</code>	Sets <code>START</code> option for XEQ files
<code>STATS</code>	Sets statistics printing option

`value` is the new value of the option. Global option values are as follows:

<code>AUTOPAD</code>	Setting <code>AUTOPAD ON</code> causes trailing spaces to be retained when operating with fixed length character strings. When <code>AUTOPAD</code> is <code>OFF</code> , the default, trailing spaces are stripped from fixed length character strings. (The <code>PAD</code> function may be used to retain the spaces.)
----------------------	--



**COMMITRATE** value must be an integer greater than or equal to 0. The default value of **COMMITRATE** is 1. Sets the frequency with which Warehouse does a commit to the databases. By default, Warehouse does a "commit" after every "transaction." Setting the **COMMITRATE** to value causes Warehouse to do a commit after every value transactions.

Setting the value of **COMMITRATE** to values greater than 1 usually increases performance but may require additional database log file storage.

Setting the value of **COMMITRATE** to 0 causes automatic commits to be suppressed. When the commit rate is 0, Warehouse only does a commit when a **COMMIT** statement is encountered and at the successful completion of the script.

A "transaction" is defined as one record selected from the outermost **READ** statement.

**MSG** value must be either **OFF** or **ON**. If set to **OFF**, no messages are displayed for data type conversions during record assignment. If set to **ON**, the default, Warehouse

	indicates missing fields and data type conversions for COPY statements and record assignments using SETVAR.
PAGELength	value must be an integer value greater than or equal to 0. A PAGELength of 0 disables page breaks between pages. The default PAGELength is 55.
PAGEWidth	value must be an integer value greater than 0. The default PAGEWidth is 80.
PRINTNULL	<p>changes the value displayed when a null value is printed. The syntax is:</p> <pre>SET PRINTNULL "&lt;null-print-value&gt;"</pre> <p>&lt;null-print-value&gt; is the string displayed when a null is printed with the PRINT statement. By default, the value is \$NULL.</p> <p>Example:</p> <pre>SET PRINTNULL "(null)" DEFINE S1, S2 : ORACLE     VARCHAR2(10) ALLOW     NULLS SETVAR S1 = "Taurus" PRINT S1, S2, S1    S2 GO Taurus    (null)    (null)</pre>
PROGRESS	value must be an integer greater than 0. Sets the frequency in seconds with which Warehouse displays progress messages during script execution. When

progress messages are enabled, Warehouse displays the number of records read, written (copied), updated and deleted every `value` seconds. The default value of `PROGRESS` is 0, indicating progress messages are not to be displayed.

`START` `value` must be either `OFF` or `ON`. If set to `OFF`, the default, Warehouse displays each line from an `XEQ` file as it is processed. If set to `ON`, lines from an `XEQ` file are not displayed, like when using the `START` statement. Display of lines can also be suppressed by running Warehouse with the `-start` option.

`STATS` `value` must be either `OFF` or `ON`. If set to `ON`, the default, statistics about the access of each file or table are displayed after processing the script is complete. If set to `OFF`, no statistics are displayed. Stats can also be suppressed by running Warehouse with the `-nostats` option.

`db-tag` specifies the database tag to which the `SET` statement is to apply.

`ARRAYIFY` Interprets adjacent, consecutively-numbered columns of the same type to be accessed with array logic.

`DEFER` Delays database writes.

`CHARSET` Assigns a character set to a

	database.
LABELS	Reports column labels on DB2/400 systems. Typically accessed only through DataBridger Studio.
RECNUMS	controls the addition of a virtual column named \$RECNUM to be appended to each record in an IMAGE database. The column holds the record number.
SHOWSQL	Controls the display of generated SQL DML for the database.

value is the new value of the option. db-tag values are as follows:

ARRAYIFY	value must be either OFF or ON. SQL (Oracle and ODBC) table layouts may now be interpreted to contain arrays. SET ARRAYIFY must be issued after the OPEN and before any other statements access the database.
DEFER	value must be either OFF or ON. Causes database writes to be deferred.
CHARSET	value must be a valid character set name enclosed in double quotes. When a database is assigned a character set, all character items from the database will be interpreted as belonging to the specified character set.

The syntax is: SET db-tag CHARSET "charset-name".

The character set name, `charset-name`, must be enclosed in quotation marks.

`SET CHARSET` must be issued after the `OPEN` but *\*before\** any other statements access the database.

`LABELS` value must be `ON`, `OFF`, or `EBCDIC`. `ON` causes Warehouse to retrieve column labels for each table. `OFF` causes Warehouse not to retrieve column labels. `EBCDIC` retrieves labels and translates them from `EBCDIC` to `ASCII`.

`RECNUMS` value must be `ON` or `OFF`. When enabled, a virtual column named `$RECNUM` is appended to each `IMAGE` database record. This record number may be used to read a record by referring to `$RECNUM`. `SET` activates record number access for the entire database, and must be specified after the related `OPEN` statement. To enable record number access just within the context of a given `READ` statement, see [Chapter Four](#), **IMAGE File Types: Read**.

`SHOWSQL` value must be `ON` or `OFF`. When enabled, tells the Warehouse Client to display the SQL DML generated on behalf of a Warehouse I/O operation.

**Considerations**

`db-option` specifies the name of the file option to be changed. `db-option` is specific to the file type. See [Chapter Four, File Types](#), for what values are permitted for each file type.

**Examples**

The `SET` statement is executed immediately and is not executed as part of running the script.

Example 1

```
SET PAGELENGTH 55
```

Sets the page length to 55 lines, which causes a page break to be printed after every 55 print lines.

Example 2

```
OPEN TDB IMAGE TESTDB PASS=MYPASS MODE=3  
SET TDB DEFER ON
```

This example opens the `IMAGE` database `TESTDB` using a database tag of `TDB`, a password of `MYPASS` and an open mode of 3, for exclusive access. The `SET` statement is used to turn `DEFER` mode on, causing database writes to be deferred by `IMAGE`.

Example 3

```
SET AUTOPAD ON  
DEFINE C1 : CHAR(8)  
DEFINE C2 : SQL CHAR(6)  
SETVAR C1 = "CHAR1"  
SETVAR C2 = "C2"  
PRINT LEN(C1)  
PRINT LEN(C2)  
PRINT C1 || C2  
GO  
8  
6  
CHAR1    C2
```

Setting `AUTOPAD ON` in this example causes operations with the fixed length characters strings `C1` and `C2` to retain their trailing spaces. If `AUTOPAD` was off, the script would have printed 5,

2, and CHAR1C2.

#### Example 4

```
OPEN TDB IMAGE TESTDB PASS=MYPASS MODE=3
OPEN ODB REMOTE UNXSYS &
  USER=REMUSR PASS=REMPAS &
  ORACLE SCOTT/TIGER &
  SID=ORADB HOME=/users/home/ora
SET COMMITRATE 400
READ OM = TDB.ORDERS FOR STAT = "CLX"
  COPY OM TO ODB.ORDERS
  READ OL = TDB.ORD-LINES &
    FOR ORDNO = OM.ORDNO
    COPY OL TO ODB.ORD_LINES
  ENDREAD
ENDREAD
```

This example opens a local IMAGE database TESTDB. A remote Oracle database on the system UNXSYS is also opened. A SET statement is used to set the commit rate to 400. This causes Warehouse to do a commit after every 400 outermost READ statements have executed. In this case 400 ORDERS records and their associated ORD-LINES records are read from the IMAGE database and copied to the remote Oracle database between commits.

#### Example 5

```
OPEN TDB IMAGE TESTDB PASS=MYPASS MODE=3
OPER ODB REMOTE UNXSYS &
  USER=REMUSR PASS=REMPAS &
  ORACLE SCOTT/TIGER &
  SID=ORADB HOME=/users/home/ora
SET COMMITRATE 400
SET PROGRESS 30
READ OM = TDB.ORDERS FOR STAT = "CLX"
  COPY OM TO ODB.ORDERS
  READ OL = TDB.ORD-LINES &
    FOR ORDNO = OM.ORDNO
    COPY OL TO ODB.ORD_LINES
  ENDREAD
ENDREAD
GO
10:21:33 Read: 3076 Write: 3075 Upd: 0 Del: 0
10:22:04 Read: 6212 Write: 6211 Upd: 0 Del: 0
```

This is the same as example 4, except that a `SET PROGRESS 30` statement has been added. After the `GO` statement progress messages are displayed as Warehouse processes the records.

### Example 6

Given an input table with the following successively numbered columns:

```
DIVNM      : CHAR(50)
QTR_[01]   : ODBC DECIMAL(16,2)
QTR_[02]   : ODBC DECIMAL(16,2)
QTR_[03]   : ODBC DECIMAL(16,2)
QTR_[04]   : ODBC DECIMAL(16,2)
```

Warehouse can be used to access the columns using array logic. With `ARRAYIFY` the columns become internally defined as:

```
QTR_ : ARRAY[1..4] OF ODBC DECIMAL(16,2)
```

The following logic will access them as an array:

```
OPEN SRCDB ODBC MYFINANCIAL
SET  SRCDB ARRAYIFY ON

DEFINE IX : ODBC NUMERIC

READ R = SRCDB.FINHIST
  PRINT "Division Name:", DIVNM
  SETVAR IX = 1

  WHILE IX <= 4
    PRINT "QTR ", IX, "=", QTR_[IX]
    SETVAR IX = IX + 1
  ENDWHILE

ENDREAD
```

This script opens an ODBC data source called `MYFINANCIAL`, and turns on `ARRAYIFY`. A counter named `IX` is defined and will be used to loop through the array of quarters. As records are read, the name is printed, then a `WHILE` loop is used to access all four columns and print their contents.



Example 7

```
SET PRINTNULL "(null)"
DEFINE S1, S2 : ORACLE VARCHAR2(10)
    ALLOW NULLS
SETVAR S1 = "Taurus"
PRINT S1, S2, S1 || S2
GO
Taurus      (null)      (null)
```

S2 is allowed to have no value because of the `ALLOW NULLS` on the `DEFINE` statement. Normally when printed, it would display nothing, but the use of the `PRINTNULL` command forces Warehouse to display the provided value.

Example 8

```
OPEN SRCDB IMAGE ...
SET SRCDB CHARSET "ANSI_X3.4-1968"
```

This example opens an `IMAGE` data source with the name `SRCDB`, then assigns the extended character set `"ANSI X3.4-1968"` to the data source.

`SET CHARSET` must be issued after the `OPEN` but *\*BEFORE\** any other statements access the database.

**SETVAR****Sets the value of a variable.**

The SETVAR statement assigns the result of an expression to a variable.

**Syntax**

```
SETVAR var-name = expression
```

var-name specifies the name of the variable. This variable must have been previously defined with the DEFINE statement.

expression is the new value of var-name. For more information on expressions, see [Chapter Five, Expressions](#).

**Considerations**

The variable referenced by var-name must have been previously defined using the DEFINE statement.

Warehouse attempts to store expression into var-name regardless of the data types involved. This may require a conversion that results in an error. For example, it is legal to use SETVAR to store a string into a number, but the string must contain a value that can be converted, e.g. "-5.62" can be converted without error, but "PN352" cannot.

**Examples**Example 1

```
DEFINE NUMFOUND : I2  
SETVAR NUMFOUND = 0
```

This example defines a new variable NUMFOUND as a four byte integer. The next statement initializes NUMFOUND to the value 0.

Example 2

```
SETVAR CUST.COUNTRY = "USA"
```

This example sets value of the field COUNTRY within the record CUST to USA.

Example 3

```
SETVAR RUNTOT[1] = MTD[1]
SETVAR IX = 2
WHILE IX <= 12
    SETVAR RUNTOT[IX]= &
        RUNTOT[IX - 1]+ MTD[IX]
    SETVAR IX = IX + 1
ENDWHILE
```

This example calculates an array of running totals into RUNTOT. The running totals are calculated from the MTD array by setting the first element of RUNTOT equal to the first element of MTD. A WHILE loop is then used with an index variable IX to calculate each successive element of RUNTOT by adding the previous RUNTOT with the current MTD indexed by IX.

**SHOW****Displays specific information.**

The `SHOW` statement displays information about an object or about global options.

**Syntax**`SHOW`

or

```
SHOW db-tag [table-name || DRIVER ||  
            [xml-table [ELEMENTS] [TABLES]  
            [DTD] ] ]
```

or

`SHOW var-name`

or

`SHOW fmt-name`

`db-tag` specifies the database tag of the database or file for which the options are to be shown.

`table-name` specifies the name of a table within the database specified by `db-tag`. When `table-name` is specified, all fields (columns) within the specified table are displayed along with their data type.

`DRIVER` displays driver information about the ODBC database specified by `db-tag`.

`xml-table` Displays the layout of "table" within the XML file.

`ELEMENTS` Displays the elements (columns) within the XML file.

`TABLES` Displays the "tables" within the XML file.

`DTD` Displays the XML DTD for the file.

`var-name` specifies the name of a variable to be described. The variable must have been previously defined with the `DEFINE` statement. If `var-name`

is a record, all fields within `var-name` are displayed along with their data type.

`fmt-name` specifies the name of a format to be described. The variable must have been previously defined with the `FORMAT` statement. If `fmt-name` is a record, all fields within `fmt-name` are displayed along with their data type.

## Considerations

The `SHOW` statement is executed immediately and is not executed as part of running the script.

If no parameters are given to `SHOW`, all global options are shown including the Warehouse version number.

The `SHOW` statement is used to display information about a specific item. The `LIST` statement may be used to obtain a list of items that may be shown.

## Examples

### Example 1

```
SHOW
```

```
AUTOPAD      : OFF
COMMITRATE   : 1
MSG          : ON
PAGELENGTH  : 55
PAGEWIDTH    : 80
PROGRESS     : 30
START        : OFF
STATS        : ON
VERSION      : Warehouse 3.02.6020-W - Oct 2 2008 10:21:07
PRINTNULL    : "$NULL"
```

Shows the global options.

### Example 2

```
OPEN TDB IMAGE TESTDB PASS=MYPASS MODE=3
SHOW TDB COMPANY
```

Dataset Name	Type	Entries	Capacity
COMPANY	D	6415	8000

Dataitem Name	Type
COMPANY-KEY	X8 Search item
MAJOR-COMPANY	X6
COMPANY-NAME	X40

.

.

.

This example opens the IMAGE database TESTDB using a database tag of TDB, a password of MYPASS and an open mode of 3. The SHOW statement is used to display the layout of the dataset (table) COMPANY within TESTDB.

### Example 3

```
DEFINE TOTAL : INTEGER
SHOW TOTAL
```

Variable definition for TOTAL:

```
INTEGER
```

This example defines the variable TOTAL as an integer. The SHOW statement is used to display the data type of TOTAL.

### Example 4

```
OPEN MYDB ODBC MSACCESSDB
SHOW MYDB DRIVER
```

Displays the following driver information for the MS Access database named MYDB

```
SQL_DBMS_NAME = ACCESS
SQL_DRIVER_NAME = odbcjt32.dll
Driver conformance level = Level 1
SQL conformance level = Minimum
```

**START**

**Executes the contents of a command file.**

The `START` command reads subsequent input from a file rather than from standard input. The statements are not displayed as they are processed.

**Syntax**

`START file-name`

`file-name` is the name of the file containing Warehouse statements to be executed. Subsequent Warehouse statements are read from `file-name` until an end of file condition is reached.

**Considerations**

The `XEQ` and `START` commands both read and process statements from a file. The difference is that the `XEQ` statement displays the statements as they are processed, but the `START` command does not.

`START` statements may be nested. `XEQ` statements within a file processed by `START` do *not* display lines.

A file may be “started” from the command line by running Warehouse with the `-start` option.

**Examples****Example 1**

The following `FORMAT` statement is entered into a file called `COFMT`.

```
FORMAT CO-FMT
CO-NAME      : X32
CO-ADDR-1    : X32
CO-ADDR-2    : X32
CO-CITY      : X32
CO-ST-ZIP    : X32
END
```

The `FORMAT` statement may be accessed using the `START` statement as in the following script:

```
OPEN F FIXED COFILE
START COFMT
```

```
READ C = F FORMAT CO-FMT &  
      ORDER-BY CO-NAME  
      PRINT CO-NAME, CO-CITY  
ENDREAD
```



**TRY**

**Defines the beginning of a TRY recovery block.**

The TRY statement defines the beginning of a try and recovery block.

**Syntax**

```
TRY
    try-statement-list
RECOVER
    recover-statement-list
ENDTRY
```

or

```
TRY
    try-statement-list
ENDTRY
```

`try-statement-list` is a list of Warehouse statements that are executed as normal, except that if an error occurs control immediately transfers to the statements after the RECOVER statement. If no error occurs while executing the `try-statement-list`, the `recover-statement-list` is skipped and control transfers to the next statement after the ENDTRY statement.

`recover-statement-list` is a list of Warehouse statements that are executed only in the event of an error in the `try-statement-list`.

**Considerations**

If there is no RECOVER statement, an error causes control to be transferred to the next statement after the ENDTRY statement.

TRY/RECOVER blocks may be nested to provide several layers of recovery.

The ESCAPE statement may be used to generate an error condition that can be trapped by a TRY/RECOVER block.

If an error occurs and there is no TRY statement in the script, or if the error occurs outside the control of a TRY statement, Warehouse displays an error

message on `stderr` and script execution terminates. In this case the Warehouse program exits with an exit condition of 1 indicating an error. (On MPE/ix, the job control word JCW is set to FATAL.)

## Examples

### Example 1

```
OPEN CDB &
  REMOTE GIRAFFE &
  USER=ouser pass=opass &
  ORACLE SCOTT/TIGER &
  HOME=/u01/oradata/ora &
  SID=ora
DEFINE TOTAL : ORACLE NUMBER
TRY
  READ C = CDB.CUSTOMERS FOR STAT = "I"
  SETVAR TOTAL = 0
  READ T = CDB.CUST_TRANS &
    FOR CUST_NO = C.CUST_NO
    SETVAR TOTAL = TOTAL + AMT
  ENDREAD
  TRY
    UPDATE C SET TOT_AMT = TOTAL
  RECOVER
    PRINT "***Error updating", cust_no
  ENDTRY
ENDREAD // CDB.CUSTOMERS
RECOVER
  ESCAPE "**** Error during script ****"
ENDTRY
```

This example opens a remote Oracle database on the system GIRAFFE. A work variable TOTAL is defined and a TRY statement is entered that encompasses the major portion of the script. If an unrecovered error occurs during processing of the statements after the TRY, control immediately transfers to the corresponding RECOVER statements, which in this case is a single ESCAPE statement. Notice there is another TRY statement before the UPDATE statement. If an error occurs during the UPDATE statement, a separate RECOVER causes the customer number to be displayed, but the script continues processing more records.

**UPDATE****Updates the current record.**

The UPDATE statement is used to change the current record from a file that was read with a READ statement.

**Syntax**

```
UPDATE read-tag SET field-name = value  
    [, field-name = value ] [, ...]
```

`read-tag` is the name of an active read tag created with the READ statement. See the READ statement in this chapter for information on read tags.

`field-name` is the name of the field within the record to be updated.

`value` is any valid expression. For more information on expressions, see [Chapter Five, Expressions](#).

**Considerations**

The UPDATE statement may only be used on files that support record update. For more information, see [Chapter Four, File Types](#).

**Examples****Example 1**

```
OPEN CDB IMAGE CUSTDB.DATABASE  
CREATE ARCFIL ARCHIVE CUSTARCH  
READ C = CDB.CUSTOMERS FOR STATUS = "I"  
    READ T = CDB.CUST-TRANS &  
        FOR CUST-NO = C.CUST-NO  
        COPY T TO ARCFIL.CUST-TRANS  
        DELETE T  
    ENDREAD  
    UPDATE C SET &  
        STATUS = "ARCHIVED", &  
        COUNT = 0  
ENDREAD // End of READ CDB.CUSTOMERS
```

This example copies and deletes CUST-TRANS records, then updates the data items STATUS to ARCHIVED and COUNT to zero. Notice the // indicating a comment on the ENDREAD statement

**WHILE**

**Defines the beginning of a WHILE loop.**

The WHILE statement defines the beginning of a while loop.

**Syntax**

```
WHILE condition [DO]
    statement-list
ENDWHILE
```

`condition` specifies a Boolean expression to be evaluated. It may be any Boolean expression containing arithmetic and string operators. The `statement-list` is executed repeatedly in a loop as long as `condition` is TRUE. As soon as `condition` is FALSE, control transfers to the statement following the `ENDWHILE`. If `condition` is FALSE upon entry to the loop, control transfers directly to the statement after the `ENDWHILE` without executing `statement-list`. For more information on Boolean expressions, see [Chapter Five, Expressions](#).

`statement-list` is a list of Warehouse statements that may include other WHILE statements.

**Examples****Example 1**

```
SETVAR IX = 1
WHILE IX <= LEN(BUF)
    IF STR(BUF, IX, 1) = '*'
        SETVAR BUF = &
            STR(BUF, 1, IX - 1) || " " || &
            STR(BUF, IX + 1, LEN(BUF) - IX)
    ENDIF
    SETVAR IX = IX + 1
ENDWHILE
```

This example steps through the string `BUF` and converts every `*` to a blank.

**XEQ****Executes the contents of a command file.**

The XEQ command reads subsequent input from a file rather than from standard input. Statements are displayed as they are processed.

**Syntax**

XEQ file-name

file-name is the name of the command file to be executed. Subsequent Warehouse statements are read from file-name until an end of file condition is reached.

**Considerations**

XEQ statements may be nested within XEQ files.

An XEQ statement is executed automatically by Warehouse if a file name is included as a parameter when Warehouse is run. In that case, Warehouse automatically executes the file name, and does an automatic GO at the end of the script file.

The XEQ and START commands both read and process statements from a file. The difference is that the XEQ statement displays the statements as they are processed, but the START command does not.

If an XEQ statement is done within a file that had been previously STARTed or if the START option is ON, then lines from file are *not* displayed as they are processed.

**Examples**Example 1

The following FORMAT statement is entered into a file called COFMT.

```
FORMAT CO-FMT
CO-NAME      : X32  // Company name
CO-ADDR-1    : X32  // Address line 1
CO-ADDR-2    : X32  // Address line 2
CO-CITY      : X32  // City
CO-ST-ZIP    : X32  // Zip code
```

```
END // End of CO-FMT
```

The FORMAT statement may be accessed using the XEQ statement as in the following script:

```
OPEN F FIXED COFILE
XEQ COFMT
READ C = F FORMAT CO-FMT &
      ORDER-BY CO-NAME
      PRINT CO-NAME, CO-CITY
ENDREAD
```

**!****Executes a system command.**

The **!** statement passes what follows after the **!** to the operating system as a command to be immediately executed. A colon (**:**) may be used instead of the exclamation point (**!**).

**Syntax****!** command

or

**:** command

command is the system-dependent command to be executed.

**Considerations**

The **!** statement is completely operating system dependent and may function differently in different operating environments.

The **!** statement is different from the **SYSTEM** function in that the **!** statement is executed immediately, but the **SYSTEM** function is always executed as part of script execution.

The **:** statement is identical in function to the **!** statement.

**Examples**Example 1**!** LISTFILE A@,2

Immediately executes a **LISTFILE** command on the MPE/iX operating system. **LISTFILE** will probably result in an error if executed on a operating system other than MPE/iX.

Example 2**!** DIR A:\*.\*

Executes a **DIR** command.

\*

**Designates a comment.**

The \* statement instructs Warehouse to ignore the text after the asterisk. This statement is used to add comments to the script.

**Considerations**

The \* only indicates a comment when it is the first non-blank character on a line.

Comments may also be placed at the end of a statement by using two slashes (//) to denote the end of the line and beginning of the comment.

**Examples**

```
* This is a comment line.
```

Warehouse ignores the text after the \*.



## **Chapter Four**

### **File Types**

### Chapter Overview

This chapter describes in detail each of the database systems and file types supported by Warehouse. For each database, all relevant Warehouse statements and how they interact with the particular database are discussed.

**ALLBASE****Allbase file access**

This section describes the considerations for each Warehouse statement that accesses an Allbase database.

Allbase is a database from Hewlett-Packard that runs on HP3000 MPE/iX based machines.

**COPY . . . TO**

Records may only be copied to an Allbase table or updateable view.

**Examples****Example 1**

```
OPEN ARCH ARCHIVE ARCFIL
OPEN PROD ALLBASE PRODDBE
READ ORDS = ARCH.ORDS FOR DATE = 920222
    COPY ORDS TO PROD.ORDDB.ORDS
    READ LINES = ARCH.ORD-LINES &
        FOR ORDNO = ORDS.ORDNO
        COPY LINES TO PROD.ORDDB.ORD-LINES
    ENDREAD
ENDREAD
```

This example opens the archive file ARCFIL, and opens the Allbase database environment PRODDBE. All ORDS records with a DATE field equal to 920222 are read from the archive file. The ORDS records are copied into the table ORDDB.ORDS in PRODDBE. The associated ORD-LINES records are read from the archive file and copied into the table ORDDB.ORD-LINES.

**CREATE**

The CREATE statement is not supported for Allbase databases.

**DELETE**

The DELETE statement may only be used on a table or updateable view.

**Examples****Example 1**

```
OPEN DBE ALLBASE PDDBE
```

```
READ M = DBE.PARTS.MASTER &  
      FOR STATUS = "NOTREF"  
      DELETE M  
ENDREAD
```

This example opens the Allbase environment file PDDBE using a database tag of DBE. The table PARTS.MASTER is read and all parts that have NOTREF in the STATUS field are selected. All selected records are then deleted.

#### OPEN

The OPEN statement is used to access an Allbase database.

Note: Only one Allbase environment may be opened within a Warehouse script.

#### Syntax

```
OPEN db-tag ALLBASE env-file-name
```

db-tag is the database tag used to reference the database in the remainder of the script.

env-file-name is the file name of the Allbase environment file.

#### Examples

##### Example 1

```
OPEN ORD ALLBASE ORDERDBE
```

This example opens the Allbase environment ORDERDBE. The database tag ORD is used to reference this database in the remainder of the script.

#### READ

The READ statement is used to read records from Allbase tables and views.

#### Syntax

```
READ read-tag = db-tag.table-ref  
      [FOR condition]  
      [ORDER BY order-list]
```

table-ref is the name of the table or view Warehouse is to read. table-ref is one of:

```
owner.table-name
owner.view-name
table-name
view-name
```

When a FOR condition is specified, Warehouse selects only those records matching condition. If no FOR condition is specified, all records in the table or view are selected.

When ORDER BY is specified, Warehouse orders (sorts) the selected records as specified by order-list.

### Examples

#### Example 1

```
OPEN SALES ALLBASE SALESDBE.PUB.DB
READ M = SALES.CUST.MASTER &
        FOR STATUS = "CLOS" &
        ORDER BY CUST_NAME
    READ D = SALES.CUST.ORDERS &
        FOR CUST_NO = M.CUST_NO
    PRINT CUST_NO, M.CUST_NAME, ORD_AMT
ENDREAD
ENDREAD
```

This example opens the Allbase environment file SALESDBE.PUB.DB using the database tag SALES. The customer master table CUST.MASTER is read selecting the records where the STATUS field has a value of CLOS. All qualifying records from CUST.MASTER are ordered in ascending order by CUST\_NAME. For each CUST.MASTER record, records from the CUST.ORDERS table are read by CUST\_NO. For each record from CUST.ORDERS, the fields, CUST\_NO and ORD\_AMT are printed along with CUST\_NAME from the CUST.MASTER table.

### UPDATE

The UPDATE statement may only be used on a table or updateable view.

### Examples

#### Example 1

```
OPEN C ALLBASE CUSTDBE
```

```
CREATE A ARCHIVE ARFILE
READ M = C.DB.MAST FOR DATE <= 921231
  COPY M TO A.DB.MAST
  UPDATE M SET STATUS = "ARCH"
ENDREAD
```

This example opens the Allbase environment file CUSTDBE using the database tag C to access this database environment in the remainder of the script. An archive file called ARFILE is created using the database tag A. The table DB.MAST is read and records where the DATE field has a value less than or equal to 921231 are selected. All qualifying records from DB.MAST are copied to the archive file and the field STATUS is updated to the value of ARCH.

**ARCHIVE****Warehouse archive file access**

This section describes the considerations for each Warehouse statement that accesses a Warehouse archive file.

A Warehouse archive file is a serially accessed file that is similar to a database in that it may contain records from many different sources in many different formats. Archive files are platform independent. Whenever a record is written to an archive file with the `COPY` statement, Warehouse writes the format of the record along with the data. Records written to an archive file from a database may subsequently be read from the archive file in the exact same format as in the original database.

The important thing to keep in mind when accessing an archive file is that it is a serial file. New archive files are created with the `CREATE` statement and are written to with the `COPY` statement. Existing archive files are opened with the `OPEN` statement and may be read with the `READ` statement, but may *not* be changed with the `COPY`, `DELETE` or `UPDATE` statements.

The archive file format has been changed since the original version of Warehouse for MPE/iX. Warehouse is able to transparently read archive files created in the original format, but cannot write archive files in the original format.

`COPY . . . TO`

When copying a record to an archive file, the archive file must have been opened with the `CREATE` statement.

Syntax

```
COPY record TO db-tag.archive-name
    [FORMAT format-name]
```

`record` is the name of the record containing the data to be copied to the archive file.

`db-tag` is the database tag of the archive file as

specified in the CREATE statement.

archive-name is the name of the destination file as it is to be recorded in the archive file. The archive-name may be in the format of db-name.table-name. When the archive file is subsequently read, this is the name used in the READ statement to retrieve the record.

format-name is the name of a format previously defined with the FORMAT statement. When format-name is specified, the record is written to the archive file with the record format specified.

## Examples

### Example 1

```
OPEN ORD IMAGE ORDDB PASS=READER MODE=5
CREATE ARCH ARCHIVE ARCHFILE
READ M = ORD.MASTER FOR STAT = "PAID"
    COPY M TO ARCH.ORDDB.MASTER
    READ D = ORD.DETAIL FOR KEY = M.KEY
        COPY D TO ARCH.ORDDB.DETAIL
    ENDREAD
ENDREAD
```

This example opens the IMAGE database ORDDB using a password of READER and an open mode of 5 and creates a new archive file ARCHFILE. The dataset MASTER from the database ORD is read selecting records where the STAT field is PAID. The qualifying MASTER records are then copied to the ORDDB.MASTER file in the archive file. For each MASTER record copied, all associated detail records are read from DETAIL and copied to the ORDDB.DETAIL file in the archive file.

## CREATE

The CREATE statement is used to create a new archive file.

## Syntax

```
CREATE db-tag ARCHIVE file-name
```

db-tag is the database tag used to reference the archive file in the remainder of the script.



`file-name` is the file name of the archive file to be created.

**Examples****Example 1**

```
CREATE AR ARCHIVE ARFILE
```

This example creates the Warehouse archive file `ARFILE`. The database tag `AR` is used to access the archive file in the remainder of the script.

**DELETE**

The `DELETE` statement is not supported for archive files.

**OPEN**

The `OPEN` statement is used to access a previously created Warehouse archive file.

**Syntax**

```
OPEN db-tag ARCHIVE file-name
```

`db-tag` is database tag used to reference the archive file in the remainder of the script.

`file-name` is file name of the archive file. The archive file may have been created by a the `CREATE` statement in a previous script, or it may be a Warehouse 1 archive file created by the older MPE/iX version of Warehouse.

**Examples****Example 1**

```
OPEN ARCF ARCHIVE ARCHFILE
```

This example opens the Warehouse archive file `ARCHFILE`. The database tag `ARCF` is used to access the archive file in the remainder of the script.

**READ**

The `READ` statement is used to read records from an archive file. Warehouse can read any record that was copied to the archive file; however, since archive files are serial, it is important to read records in the same order they were written.

Syntax

```
READ read-tag = db-tag.archive-name
    [FORMAT format-name]
    [FOR condition]
    [ORDER BY order-list]
```

When a FORMAT is specified, format-name overrides the record definition in the archive file and uses the field names and types from format format-name.

When a FOR condition is specified, Warehouse selects only the records from the archive file that match condition.

When ORDER BY is specified, Warehouse first sorts the selected records, then processes the records in the order specified by order-list.

Examples

The following examples show how to read from an archive file that was created with the following script:

```
OPEN ORD IMAGE ORDDDB PASS=READER MODE=5
CREATE ARCH ARCHIVE ARCHFILE
* Copy CUST records first
READ F = ORD.CUST FOR STAT = "INACT"
    COPY F TO ARCH.CUST
ENDREAD
* Copy MASTER and DETAIL records
READ M = ORD.MASTER FOR STAT = "PAID"
    COPY M TO ARCH.MASTER
    READ D = ORD.DETAIL FOR ORNO = M.ORNO
        COPY D TO ARCH.DETAIL
    ENDREAD
ENDREAD
```

The above script creates an archive file called ARCHFILE that looks like the following:

```
ARCHFILE
CUST
CUST
CUST
CUST
.
```

.
MASTER
DETAIL
DETAIL
MASTER
DETAIL
DETAIL
DETAIL
MASTER
.
.
.

### Example 1

```
OPEN ARCH ARCHIVE ARCHFILE
* Read CUST records first
READ F = ARCH.CUST
  PRINT CUST-NO, CUST-NAME
ENDREAD
* Read MASTER and DETAIL records
READ M = ARCH.MASTER
  PRINT ORNO, CUST-NO
  READ D = ARCH.DETAIL
    PRINT ITEM-NO, QTY, AMT
  ENDREAD
ENDREAD
```

This example opens the archive file ARCHFILE and first reads and prints information from all CUST records. The script then reads all MASTER records and prints the ORNO and CUST-NO fields from the MASTER record. For each MASTER record all associated DETAIL records are read, with the ITEM-NO, QTY, and AMT being printed from the DETAIL records.

This script executes successfully because it was constructed to read the files CUST, MASTER and DETAIL exactly as they were originally written to the archive file.

### Example 2

```
OPEN ARCH ARCHIVE ARCHFILE
* Read MASTER records
READ M = ARCH.MASTER
```

```
PRINT ORNO, CUST-NO
ENDREAD
```

This example opens the archive file ARCHFILE, reads all MASTER records, then prints the ORNO and CUST-NO fields from each MASTER record. The CUST records and the DETAIL records are ignored by this script.

This script executes successfully because the CUST records and DETAIL records can be safely ignored as they are encountered while Warehouse reads serially through the archive file.

### Example 3 (Unsuccessful)

```
OPEN ARCH ARCHIVE ARCHFILE
* Read MASTER records
READ M = ARCH.MASTER
  PRINT ORNO, CUST-NO
ENDREAD
* Read DETAIL records
READ D = ARCH.DETAIL
  PRINT ITEM-NO, QTY, AMT
ENDREAD
```

This example opens the archive file ARCHFILE and first attempts to read all MASTER records and print from the MASTER record. It then attempts to read all the DETAIL records. This script will not produce the desired results because the DETAIL records are intermixed with the MASTER records on the archive file, but this script assumes the DETAIL records are separate from the MASTER records.

SET

There are no special SET options for archive files.

UPDATE

The UPDATE statement is not supported for archive files.

## CSV

**CSV (Comma Separated Values) file access**

This section describes the considerations for each Warehouse statement that accesses comma separated value files.

The type `csv` is used to designate all files that have delimiter separated values in a text file. The delimiter is not required to be a comma. When reading, Warehouse separates all fields by the delimiter; when writing fields are stripped of leading and trailing spaces and separated by the delimiter. Separated values are used commonly to move data between applications because many different applications support them.

CSV files should only be used when reading or writing character data. When reading or writing binary data, a `FIXED` file should probably be used.

## COPY...TO

Leading and trailing spaces are stripped from each field before it is written to the CSV file and the delimiter is used to separate each field. Fields which contain the delimiter character are surrounded by quotation marks.

When copying to a CSV file, the file must have been opened for write access. This is typically done with the `CREATE` statement.

## Examples

Example 1

```
OPEN ARCH ARCHIVE ARCFIL
CREATE ORFIL CSV ORDFILE
READ ORDS = ARCH.ORDS FOR DATE = 920222
  COPY ORDS TO ORFIL
ENDREAD
```

This example opens the archive file `ARCFIL`, and creates a new CSV file `ORDFILE` which is given a database tag `ORFIL`. All `ORDS` records from the archive file with a `DATE` field equal to 920222 are read and are then copied into the file `ORDFILE`.

## CREATE

The `CREATE` statement is used to create a new CSV

file.

Syntax

```
CREATE file-tag CSV file-name
    [MODE=mode]
    [DELIM=delimiter-character]
    [QUOTE=quotation-character]
    [ESCAPE=escape-character]
    [STRIP=strip-character]
    [MAXREC=max-recsize]
    [ALLQUOTED]
    [FIELDNAMES]
```

`file-tag` is the file tag used to reference the file in the remainder of the script.

`file-name` is the file name of the file to be created.

`mode` is the mode the file is to be accessed in. The mode definitions are as follows:

READ	Read access to file.
WRITE	Write access to file. ERASE or READ must be specified with WRITE. (On MPE systems, WRITE may be specified without ERASE or READ.)
APPEND	Append access to file. WRITE or ERASE not permitted with APPEND access.
BINARY	File contains BINARY data. Implementation of this parameter depends on the operating system. It is recommended whenever accessing a file with binary data.
ERASE	Erases contents of file prior to access. If ERASE is specified, file is not required to already exist. If erase is not specified, file is required to already exist.
COMMIT	(Windows only.) Contents of the file buffer are written directly to

	disk when <code>fflush()</code> is called.
VAR	(MPE only) Do not internally buffer records. Assume each read and write consists of one record. This is necessary to read variable length records.
EXCLUSIVE	(MPE and Warehouse message files only) Access the file exclusively.
SHARE	(MPE only) Access the file in shared mode.
LOCK	(MPE only) Access the file with locking enabled.
TRANS	Access the file in transaction protected mode. Using TRANS allows COMMIT and ROLLBACK by keeping all file changes in memory buffers until a commit or rollback is done. A commit writes the buffers to disk, and a rollback discards the buffers.
MSG	File is message file. On MPE systems, this is an MPE message file. On other platforms, this is a Warehouse message file.
NDR	File is an MPE message file to be read with non-destructive reads. On MPE, specifying NDR implies MSG too.
CLIB	(MPE only) Use C library instead of MPE intrinsic calls to access files.
UPDATE	(MPE only) Access the file with update access. Use of UPDATE is necessary to delete or update KSAM files.

If mode is not specified, the default mode of `w` for write access is used.

`delimiter-character` is a single character that specifies how the fields are to be delimited in the output. If the delimiter is a space, it needs to be enclosed in quotation marks. The default `delimiter-character` is a comma (,).

`quotation-character` is a single character that specifies how fields that contain the `delimiter-character` or `quote-character` are grouped into a single field. When a field that contains either the `delimiter-character` or the `quote-character` is written, it is enclosed in the `quote-character`. The default `quote-character` is a double quote (").

`escape-character` determines how the `quote-character` is handled in a quoted field. Fields that contain the `quote-character` are output surrounded by the `quote-character` and each time the `quote-character` actually occurs in the field, it is preceded by the `escape-character`. The default `escape-character` is a double quote (").

`strip-character` indicates a character that is stripped at the beginning and end of each field. The default `strip-character` is a space. To indicate no characters are stripped, use `STRIP=" "`.

`max-recsize` indicates the maximum size of records written to the file. Records longer than `max-recsize` characters are truncated. The default `max-recsize` is 1024.

`ALLQUOTED` indicates that *all* fields are to be enclosed in the `quotation-character` when output. By default, fields are only enclosed by the `quotation-character` if needed.



FIELDNAMES indicates that the first record output contains a list of field names separated by the delimiter-character. By default, the field names are not output to the first record.

### Examples

The following examples assume CUST records are output to the file CUSTFIL. The CUST records are:

CUSTNO	CUSTNAME	BALANCE
88425	Big Apple Orchards	449.60
92010	Green, Field , & Summers	1195.32
90028	Dandy Daisy Farm	4402.79
94008	Circle "T" Tractors	24017.08

### Example 1

```
OPEN DB ORACLE ...
CREATE CF CSV CUSTFIL
READ C = DB.CUST
  COPY C TO CF
ENDREAD
```

The CREATE statement results in CUSTFIL that looks as follows:

```
88425,Big Apple Orchards,449.60
92010,"Green, Field , & Summers",1195.32
90028,Dandy Daisy Farm,4402.79
94008,"Circle ""T"" Tractors",24017.08
```

### Example 2

```
OPEN DB ORACLE ...
CREATE CF CSV CUSTFIL &
  MODE=READ WRITE ERASE &
  DELIM=; QUOTE=" ' " FIELDNAMES
READ C = DB.CUST
  COPY C TO CF
ENDREAD
```

The CREATE statement results in CUSTFIL that looks as follows:

```
CUSTNO;CUSTNAME;BALANCE
88425;Big Apple Orchards;449.60
92010;'Green, Field , & Summers';1195.32
90028;Dandy Daisy Farm;4402.79
94008;'Circle "T" Tractors';24017.08
```

Example 3

```

OPEN DB ORACLE ...
CREATE CF CSV CUSTFIL &
    ALLQUOTED ESCAPE=\
READ C = DB.CUST
    COPY C TO CF
ENDREAD

```

The CREATE statement results in CUSTFIL that looks as follows:

```

"88425","Big Apple Orchards","449.60"
"92010","Green, Field , & Summers","1195.32"
"90028","Dandy Daisy Farm","4402.79"
"94008","Circle \"T\" Tractors","24017.08"

```

**DELETE**

The DELETE statement is not supported for CSV files.

**OPEN**

The OPEN statement is used to access an existing CSV file.

**Syntax**

```

OPEN file-tag CSV file-name
    [MODE=mode]
    [DELIM=delimiter-character]
    [QUOTE=quotation-character]
    [ESCAPE=escape-character]
    [STRIP=strip-character]
    [MAXREC=max-recsize]
    [ALLQUOTED]
    [FIELDNAMES]

```

file-tag is the file tag used to reference the file in the remainder of the script.

file-name is the file name of the CSV file to be opened.

mode is the mode the file is to be accessed in. The mode definitions are as follows:

READ            Read access to file.

WRITE	Write access to file. ERASE or READ must be specified with WRITE. (On MPE systems, WRITE may be specified without ERASE or READ.)
APPEND	Append access to file. WRITE or ERASE not permitted with APPEND access.
BINARY	File contains BINARY data. Implementation of this parameter depends on the operating system. It is recommended whenever accessing a file with binary data.
ERASE	Erases contents of file prior to access. If ERASE is specified, file is not required to already exist. If erase is not specified, file is required to already exist.
COMMIT	(Windows only.) Contents of the file buffer are written directly to disk when <code>fflush()</code> is called.
VAR	(MPE only) Do not internally buffer records. Assume each read and write is consists of one record. This is necessary to read variable length records.
EXCLUSIVE	(MPE and Warehouse message files only) Access the file exclusively.
SHARE	(MPE only) Access the file in shared mode.
LOCK	(MPE only) Access the file with locking enabled.
TRANS	Access the file in transaction protected mode. Using TRANS allows COMMIT and ROLLBACK by keeping all file changes in memory

buffers until a commit or rollback is done. A commit writes the buffers to disk, and a rollback discards the buffers.

MSG	File is message file. On MPE systems, this is an MPE message file. On other platforms, this is a Warehouse message file.
NDR	File is an MPE message file to be read with non-destructive reads. On MPE, specifying NDR implies MSG too.
CLIB	(MPE only) Use C library instead of MPE intrinsic calls to access files.
UPDATE	(MPE only) Access the file with update access. Use of UPDATE is necessary to delete or update KSAM files.

If mode is not specified, the default mode of READ is used.

`delimiter-character` is a single character that specifies how the fields are delimited in the file. If the delimiter is a space, it needs to be enclosed in quotation marks. The default `delimiter-character` is a comma (,).

`quotation-character` is a single character that specifies how fields that contain the `delimiter-character` or `quote-character` are grouped into a single field. When a field begins with the `quote-character`, all characters between it and the corresponding close `quote-character` are considered part of the field. To indicate the `quote-character` within a field, the `escape-character` must precede the `quote-character`. The default `quote-character` is a double quote (").

`escape-character` determines how the `quote-`

character is handled in a quoted field. Inside a quoted field, the escape-character is skipped and the next character is included in the field. The default escape-character is a double quote (").

strip-character indicates a character that is stripped at the beginning and end of each field. The default strip-character is a space. To indicate no characters are stripped, use STRIP=" ".

max-recsize indicates the maximum size of records read from the file. Records longer than max-recsize characters are truncated. The default max-recsize is 1024.

ALLQUOTED indicates that *all* fields should be enclosed in the quotation-character when read. This specification is not enforced when reading a CSV file.

FIELDNAMES indicates that the first record output contains a list of field names separated by the delimiter-character. When FIELDNAMES is specified, the first record is read immediately and the fields of the CSV file are determined based on the first record. Subsequent READ and other statements use the field names from the first record. If FIELDNAMES is not specified, the READ statement must contain a FORMAT.

### Examples

The following examples assume CUST records are output to the file CUSTFIL. The CUST records are:

<u>CUSTNO</u>	<u>CUSTNAME</u>	<u>BALANCE</u>
88425	Big Apple Orchards	449.60
92010	Green, Field , & Summers	1195.32
90028	Dandy Daisy Farm	4402.79
94008	Circle "T" Tractors	24017.08

### Example 1

To read a CSV file called CUSTFIL that looks like the following:

```
88425,Big Apple Orchards,449.60
92010,"Green, Field , & Summers",1195.32
90028,Dandy Daisy Farm,4402.79
94008,"Circle ""T"" Tractors",24017.08
```

Use a script that looks like:

```
OPEN CF CSV CUSTFIL
FORMAT CUSTFMT
  CUSTNO : ORACLE NUMBER(5)
  CUSTNAME : ORACLE CHAR(40)
  BALANCE : ORACLE NUMBER(8,2)
END
READ C = CF FORMAT CUSTFMT
.
.
.
ENDREAD
```

### Example 2

To read a CSV file called CUSTFIL that looks like the following:

```
CUSTNO;CUSTNAME;BALANCE
88425;Big Apple Orchards;449.60
92010;'Green, Field , & Summers';1195.32
90028;Dandy Daisy Farm;4402.79
94008;'Circle "T" Tractors';24017.08
```

Use a script that looks like:

```
OPEN CF CSV CUSTFIL &
  MODE=READ WRITE ERASE &
  DELIM=; QUOTE="'" FIELDNAMES
READ C = CF
.
.
.
ENDREAD
```

Notice that since `FIELDNAMES` is specified, no `FORMAT` is used on the `READ` statement.

READ

The `READ` statement is used to read records from CSV files.

## Syntax

```
READ read-tag = db-tag  
    [FORMAT format-name]  
    [FOR condition]  
    [ORDER BY order-list]
```

FORMAT is used to specify the record format of the file. FORMAT is required to specify the record layout of CSV files, except when the file is opened with the FIELDNAMES option.

When a FOR condition is specified, Warehouse selects only those records matching condition. If a FOR condition is not specified, all records in the file are selected.

When ORDER BY is specified, Warehouse orders the records as specified by order-list.

## Examples

Example 1

```
OPEN SALES CSV SALESFIL &  
    FIELDNAMES DELIM=/ QUOTE=" "  
OPEN SDB IMAGE SALES.DB PASS=MYPASS  
READ F = SALES  
    READ C = SDB.ORDERS &  
        FOR CO-NUM = F.CUSTNO  
        PRINT CUST-NO, F.CUSTNAME, ORD-AMT  
    ENDREAD  
ENDREAD
```

This example opens the CSV file SALESFIL using the database tag SALES and the mode of r for read access. The IMAGE database SALES.DB is also opened using a database tag of SDB and a password of MYPASS. The file SALESFIL looks as follows:

```
CUSTNO/CUSTNAME/BALANCE  
88425/Big Apple Orchards/449.60  
92010/Green, Field, & Summers/1195.32  
90028/Dandy Daisy Farm/4402.79  
94008/Circle "T" Tractors/24017.08
```

Matching orders are then read from the ORDERS dataset in SALES.DB. For each record from CUST.ORDERS, the fields CUST-NO and ORD-AMT are printed along with CUSTNAME from the

SALESFIL file.

SET

There are no special SET options for CSV files.

UPDATE

The UPDATE statement is not supported for CSV files.



## DB2

## DB2 file access

This section describes the considerations for each Warehouse statement that accesses a DB2 database. DB2 is a database from International Business Machines Corporation (IBM).

When accessing a DB2 database Warehouse uses SQL data types.

## COPY...TO

Records may only be copied to a DB2 table or updateable view.

## Syntax

`COPY record TO output-table`

`record` is the name of a record created with either the `DEFINE` statement or a read tag created by the `READ` statement.

`output-table` is the name of the DB2 table to which the record is copied.

## Examples

Example 1

```
OPEN ARCH ARCHIVE ARCFIL
OPEN PROD DB2 ORDERS
READ ORDS = ARCH.ORDS FOR DATE = 920222
  COPY ORDS TO PROD.ORDDB.ORDS
  READ LINES = ARCH.ORD_LINES &
    FOR ORDNO = ORDS.ORDNO
    COPY LINES TO PROD.ORDDB.ORDLINES
  ENDREAD
ENDREAD
```

This example opens the archive file `ARCFIL`, and opens the DB2 database `ORDERS`. All `ORDS` records with a `DATE` field equal to `920222` are read from the archive file. The `ORDS` records are then copied into the table `ORDDB.ORDS` in the DB2 database. The associated `ORD_LINES` records are then read from the archive file and copied into the table `ORDDB.ORDLINES`.

## CREATE

The `CREATE` statement is not supported for DB2

databases.

## DELETE

The DELETE statement may only be used on a table or updateable view. The DELETE statement may not be used if the records were read using ORDER BY.

## Examples

### Example 1

```
OPEN DB DB2 MYDB USER=MYUSER PASS=MYPASS
READ M = DB.PARTS.MASTER &
    FOR STATUS = "NOTREF"
    DELETE M
ENDREAD
```

This example opens the DB2 database MYDB using a user name of MYUSER and a user password of MYPASS. The table PARTS.MASTER is read for all parts that have NOTREF in the STATUS field. All selected records are then deleted.

## OPEN

The OPEN statement is used to access a DB2 database.

## Syntax

```
OPEN db-tag DB2 data-source
    [USER=user-name]
    [PASSWORD=user-password]
    [EPASS1=encrypted-password]
    [DB2DIR=db2-dir]
    [DB2INSTANCE=db2-instance]
    [SCHEMA=schema-name]
```

db-tag is the database tag used to reference the database in the remainder of the script.

data-source is the name of the DB2 data source as it has been set up using the DB232 Control Panel.

user-name is the name of the user accessing the data source.

user-password is the password for user-name.

encrypted-password is an encrypted password for user-name on the DB2 data source. An encrypted password for use in the OPEN statement may be generated by running Warehouse with -c (See **Checking Warehouse Server Connections** in [Chapter Seven](#)) or by DataBridger Studio. Password encryption is done by a proprietary algorithm based on the Data Encryption Standard (DES).

db2-dir is the home directory of the DB2 instance. If db2-dir is not specified, Warehouse uses the DB2 environment variable called DB2DIR.

db2-instance is the DB2 instance. If db2-instance is not specified, Warehouse uses the DB2 environment variable called DB2INSTANCE.

Schema-name sets the default schema for the database. This option is usually necessary when accessing an iSeries (AS/400) DB2 database.

### Examples

#### Example 1

```
OPEN ORD DB2 MYDB2DB &
  USER=MGR PASSWORD=EAGLE
```

Warehouse is run on AIX and a DB2 data source named MYDB2DB is opened using a user name MGR and a password EAGLE. The database is assigned a Warehouse database tag of ORD to reference the database in the remainder of the script.

### READ

The READ statement is used to read records from DB2 tables and views.

### Syntax

```
READ read-tag = db-tag.table-ref
  [FOR condition]
  [ORDER BY order-list]
```

table-ref is the name of the table or view Warehouse is to read. table-ref is one of:

```
owner.table-name
owner.view-name
```

table-name  
view-name

When a FOR condition is specified, Warehouse selects only those records matching condition. If no FOR condition is specified, all records in the table or view are selected.

When ORDER BY is specified, Warehouse orders (sorts) the records as specified by order-list.

NOTE: When ORDER BY is specified, the records may not be deleted or updated.

### Examples

#### Example 1

```
OPEN SALES DB2 SALEINFO
READ M = SALES.CUST.MASTER &
        FOR STATUS = "CLOS" &
        ORDER BY CUST_NAME
    READ D = SALES.CUST.ORDERS &
        FOR CUST_NO = M.CUST_NO
    PRINT CUST_NO, M.CUST_NAME, ORD_AMT
ENDREAD
ENDREAD
```

This example opens the DB2 data source called SALEINFO. The database tag SALES is used to access this database environment in the remainder of the script. The customer master table CUST.MASTER is read and records where the STATUS field has a value of CLOS are selected. All qualifying records from CUST.MASTER are ordered by the value in the CUST\_NAME field. For each CUST.MASTER record, detail records from the CUST.ORDERS table are read using the indexed field CUST\_NO. For each record from CUST.ORDERS, the fields, CUST\_NO and ORD\_AMT are printed along with CUST\_NAME from the CUST.MASTER table.

### SET

The SET statement is used to set DB2 access options.

## Syntax

SET db-tag DB2-option value

db-tag specifies the database tag of the DB2 database to which the SET statement is to apply.

DB2-option specifies the name of the option to be changed. DB2-option must be one of the following for DB2 databases:

LONGSIZE	Sets the maximum length of the LONG data types. When reading from or writing to a DB2 database, LONGSIZE controls the maximum number of characters (bytes) that a LONG VARBINARY or LONG VARCHAR field may contain. The default value of LONGSIZE is 10,000.
----------	--

## Examples

Example 1

```
OPEN C DB2 XDB
SET C LONGSIZE 250000
```

This example opens a DB2 data source called XDB. The database tag C is used to access the database in the remainder of the script. The LONGSIZE is set to 250,000 allowing up to 250,000 characters to be read into or written from LONG and LONG RAW fields.

## UPDATE

The UPDATE statement may only be used on a table or updateable view. The UPDATE statement may not be used if the records were read using ORDER BY.

## Examples

Example 1

```
OPEN C DB2 MFIL USER=JONES PASS=AAA
CREATE A ARCHIVE ARFILE
READ M = C.DB.MAST FOR STATUS = "CLOS"
    COPY M TO A.DB.MAST
    UPDATE M SET STATUS = "ARCH"
ENDREAD
```

This example opens a DB2 data source called `MFIL` using a user name of `JONES` and a user password of `AAA`. The database tag `C` is used to access the database in the remainder of the script. An archive file called `ARFILE` is created using the database tag `A`. The table `DB.MAST` is read and records where the `STATUS` field has a value equal to `CLOS` are selected. All qualifying records from `DB.MAST` are copied to the archive file and the field `STATUS` is updated to the value of `ARCH`.

**FIXED****Fixed length record file access**

This section describes the considerations for each Warehouse statement that accesses fixed length record files.

Fixed length record file access is specified using the type `FIXED` in the `OPEN` or `CREATE` statement and is supported on all Warehouse platforms.

Fixed length files should be used primarily when reading or writing binary data. When reading or writing character data, a `TEXT` file should probably be used.

**COPY...TO**

When copying to a fixed record length file, the file must have been opened for write access. This is typically done with the `CREATE` statement.

**Examples****Example 1**

```
OPEN ARCH ARCHIVE ARCFIL
CREATE ORFIL FIXED ORDFILE
READ ORDS = ARCH.ORDS FOR DATE = 920222
  COPY ORDS TO ORFIL
ENDREAD
```

This example opens the archive file `ARCFIL`, and creates a new fixed record length file `ORDFILE` which is given a database tag `ORFIL`. All `ORDS` records from the archive file with a `DATE` field equal to 920222 are read and are then copied into the file `ORDFILE`.

**CREATE**

The `CREATE` statement is used to create a new fixed record length file.

**Syntax**

```
CREATE file-tag FIXED file-name
      [MODE=mode]
```

`file-tag` is the file tag used to reference the file in the remainder of the script.

`file-name` is the file name of the file to be created.

mode is the mode the file is to be accessed in. The mode definitions are as follows:

READ	Read access to file.
WRITE	Write access to file. ERASE or READ must be specified with WRITE. (On MPE systems, WRITE may be specified without ERASE or READ.)
APPEND	Append access to file. WRITE or ERASE not permitted with APPEND access.
BINARY	File contains BINARY data. Implementation of this parameter depends on the operating system. It is recommended whenever accessing a file with binary data.
ERASE	Erases contents of file prior to access. If ERASE is specified, file is not required to already exist. If erase is not specified, file is required to already exist.
COMMIT	(Windows only.) Contents of the file buffer are written directly to disk when <code>fflush()</code> is called.
VAR	(MPE only) Do not internally buffer records. Assume each read and write is consists of one record. This is necessary to read variable length records.
EXCLUSIVE	(MPE and Warehouse message files only) Access the file exclusively.
SHARE	(MPE only) Access the file in shared mode.
LOCK	(MPE only) Access the file with



locking enabled.

TRANS	Access the file in transaction protected mode. Using TRANS allows COMMIT and ROLLBACK by keeping all file changes in memory buffers until a commit or rollback is done. A commit writes the buffers to disk, and a rollback discards the buffers.
MSG	File is message file. On MPE systems, this is an MPE message file. On other platforms, this is a Warehouse message file.
NDR	File is an MPE message file to be read with non-destructive reads. On MPE, specifying NDR implies MSG too.
CLIB	(MPE only) Use C library instead of MPE intrinsic calls to access files.
UPDATE	(MPE only) Access the file with update access. Use of UPDATE is necessary to delete or update KSAM files.

If mode is not specified, the default mode of WRITE is used. On non-Unix systems, appending BINARY to the mode is used to indicate a binary file. On Unix systems, this has no effect but is permitted for compatibility. See the OPEN statement for more information on mode.

## Examples

### Example 1

```
CREATE ORD FIXED ORDFILE
```

This example creates the fixed length record file ORDFILE for write access using the database tag ORD to access the file in the remainder of the script.

Example 2

```
CREATE TMP FIXED TEMP &  
      MODE=READ WRITE ERASE
```

This example creates the fixed length record file TEMP for read and write access using the database tag TMP to access the file in the remainder of the script.

## DELETE

The DELETE statement is not supported for fixed record length files.

## OPEN

The OPEN statement is used to access an existing fixed record length file.

## Syntax

```
OPEN file-tag FIXED file-name  
      [MODE=mode] [NDR | MSG]
```

file-tag is the file tag used to reference the file in the remainder of the script.

file-name is the file name of the fixed length record file to be opened.

mode is the mode in which the file is accessed. The mode definitions are as follows:

READ	Read access to file.
WRITE	Write access to file. ERASE or READ must be specified with WRITE. (On MPE systems, WRITE may be specified without ERASE or READ.)
APPEND	Append access to file. WRITE or ERASE not permitted with APPEND access.
BINARY	File contains BINARY data. Implementation of this parameter depends on the operating system.

---

	It is recommended whenever accessing a file with binary data.
ERASE	Erases contents of file prior to access. If <code>ERASE</code> is specified, file is not required to already exist. If erase is not specified, file is required to already exist.
COMMIT	(Windows only.) Contents of the file buffer are written directly to disk when <code>fflush()</code> is called.
VAR	(MPE only) Do not internally buffer records. Assume each read and write is consists of one record. This is necessary to read variable length records.
EXCLUSIVE	(MPE and Warehouse message files only) Access the file exclusively.
SHARE	(MPE only) Access the file in shared mode.
LOCK	(MPE only) Access the file with locking enabled.
TRANS	Access the file in transaction protected mode. Using <code>TRANS</code> allows <code>COMMIT</code> and <code>ROLLBACK</code> by keeping all file changes in memory buffers until a commit or rollback is done. A commit writes the buffers to disk, and a rollback discards the buffers.
MSG	File is message file. On MPE systems, this is an MPE message file. On other platforms, this is a Warehouse message file.
NDR	File is an MPE message file to be read with non-destructive reads.

On MPE, specifying NDR implies MSG too.

CLIB (MPE only) Use C library instead of MPE intrinsic calls to access files.

UPDATE (MPE only) Access the file with update access. Use of UPDATE is necessary to delete or update KSAM files.

If mode is not specified, the default mode of READ WRITE for read and write access is used. On non-Unix systems, appending BINARY to the mode is used to indicate a binary file. On Unix systems, this has no effect but is permitted for compatibility.

MPE/iX Files : By default data read from or written to a FIXED file is buffered by Warehouse without regard to MPE/iX record boundaries. The size of the FORMAT on the READ statement determines the number of bytes to read. For example, if the format is 40 bytes, but the MPE/iX record size is 80 bytes, the first Warehouse record will be the first half on MPE record 1, the second Warehouse record will be the second half MPE record 1 and so on. On the other hand if the format is 80 bytes and the MPE record size is 40 bytes, the first Warehouse record will be the MPE records 1 and 2.

Warehouse can also perform file operations on a traditional record basis. To indicate that file operations are to be done on a record basis, append VAR to the modes. This is particularly useful when reading variable record length files. To read and write MPE/iX file by record use:

MODE=VAR	Read/write access (default)
MODE=READ VAR	Read only access
MODE=WRITE VAR	Write access; discards previous contents
MODE=READ WRITE VAR	Read/write access;

discards previous  
contents  
MODE=APPEND VAR Append access

NDR is only available for MPE/iX message files. The NDR option specifies that each message file record is to be first read with a *non-destructive read*. Before the next record is read from the message file, a *destructive read* is done to remove the record from the file. Using NDR prevents any records from being lost from the message file if there is a system problem or abort during Warehouse script execution.

MSG provides access to the Warehouse message file (a capture file) – intended to duplicate the functions of message files found on the MPE/iX operating system. Message files must be open with a mode of either *r* for reading or *w* for writing.

When writing to a message file, it is always appended to the end of the file.

When reading from a message file, the keyword NDR may be used instead of MSG. This is to maintain script compatibility between MPE/iX systems and other systems.

When reading from a message file, the oldest record is returned first. Destructive reads are handled the same as with NDR.

If attempting to read from a file that is empty, Warehouse waits until a record is written if at least one other process has the file open for writing. If attempting to read from a file that is empty and no process has the file open for writing, an end of file condition is returned. There is an exception for the first read of the file. The first read of the file always waits if the file is empty. Thereafter, a read will only wait on an empty file if there is at least one other process writing to the file.

Transaction Files:	<p>FIXED or TEXT files opened with TRANS appended to the mode (e.g. "MODE=READ WRITE TRANS") support commit and rollback transaction processing. This is implemented internally in Warehouse by keeping track of all file changes in internal buffers. If the file is subsequently rolled back, all changes to the file are discarded. If the file is committed, then all changes are written to disk.</p>
Examples	<p><u>Example 1</u></p> <pre>OPEN ORD FIXED ORDFILE</pre> <p>This example opens the fixed length record file ORDFILE for read access using the database tag ORD to access the file in the remainder of the script.</p> <p><u>Example 2</u></p> <pre>OPEN TMP FIXED TEMP MODE=READ WRITE</pre> <p>This example opens the fixed length record file TEMP for read and write access using the database tag TMP to access the file in the remainder of the script.</p> <p><u>Example 3</u></p> <pre>OPEN CDRVR FIXED /home/dbadm/custdrvr &amp; MODE=READ WRITE TRANS</pre> <p>This example opens the fixed length record file custdrvr in the directory /home/dbadm. The file is opened for transaction protected read/write access. The database tag CDRVR is used to access the file in the remainder of the script.</p>
READ	<p>The READ statement is used to read records from fixed length record files.</p>
Syntax	<pre>READ read-tag = file-tag       FORMAT format-name       [FOR condition]       [ORDER BY order-list]</pre>

FORMAT is used to specify the record format of the file. FORMAT is required to specify the record layout of fixed length record files.

When a FOR condition is specified, Warehouse selects only those records matching condition. If no FOR condition is specified, all records in the file are selected.

When ORDER BY is specified, Warehouse orders (sorts) the records as specified by order-list.

## Examples

### Example 1

```
OPEN SALES FIXED SALESFIL MODE=READ
OPEN SDB IMAGE SALES.DB PASS=MYPASS
FORMAT SALESFIL_FMT
  CO-NUM : X8
  CO-NAME : X40
END
READ F = SALES FORMAT SALESFIL_FMT
  READ C = SDB.ORDERS &
    FOR CO-NUM = F.CO-NUM
    PRINT CUST-NO, F.CO-NAME, ORD-AMT
  ENDREAD
ENDREAD
```

This example opens the fixed length record file SALESFIL using the file tag SALES and the mode of r for read access. The IMAGE database SALES.DB is also opened using a database tag of SDB and a password of MYPASS. The record format SALESFIL\_FMT is defined with the two fields CO-NUM and CO-NAME. The file SALESFIL is read using the format SALESFIL\_FMT with all records being selected. Matching orders are then read from the ORDERS dataset in SALES.DB. For each record from CUST.ORDERS, the fields CUST-NO and ORD-AMT are printed along with CO-NAME from the SALESFIL file.

## SET

There are no special SET options for fixed length record files.

**UPDATE**

The UPDATE statement may only be used on a file opened for read and write access.

**Examples****Example 1**

```
OPEN CUSTS FIXED CUSTFILE &
  MODE=READ WRITE
FORMAT CUST_FMT
  CUST-NUM      : X8
  CUST-NAME     : X40
  CUST-ADDR     : X40
  CUST-CITY     : X20
  CUST-ZIP      : X10
  CUST-COUNTRY  : X20
END
READ C = CUSTS FORMAT CUST_FMT &
  FOR CUST-COUNTRY = " "
  UPDATE CUSTS SET CUST-COUNTRY = "USA"
ENDREAD
```

This example opens the fixed length record file CUSTFILE using the file tag CUSTS and the mode of READ WRITE for read and write access. The record format CUST\_FMT is defined with the fields CUST-NUM, CUST-NAME, CUST-ADDR, CUST-CITY, CUST-ZIP, and CUST-COUNTRY. CUSTFILE file is read using the format CUST\_FMT, selecting only records having blanks in the CUST-COUNTRY field. The CUST-COUNTRY is updated to the value USA.



**IMAGE****IMAGE file access**

This section describes the considerations for each Warehouse statement that accesses an IMAGE database.

IMAGE is a database system from Hewlett-Packard that runs on HP3000 MPE based machines. IMAGE is also called TurboIMAGE and IMAGE/SQL.

**COPY...TO**

When copying to an IMAGE dataset, the database must have been opened for write access.

As required by IMAGE, when copying to a detail dataset, all associated manual master records must exist.

COPY may not be used to copy records to an automatic master dataset.

**Examples****Example 1**

```
OPEN PROD IMAGE PRODDB PASS=READER MODE=5
OPEN TEST IMAGE TESTDB PASS=WRITER MODE=3
DEFINE NUM : IMAGE I2 VALUE 1
READ M = PROD.MASTER-SET FOR NUM <= 100
    COPY M TO TEST.MASTER-SET
    READ D = PROD.DETAIL-SET &
        FOR KEY = M.KEY
        COPY D TO TEST.DETAIL-SET
    ENDREAD
    SETVAR NUM = NUM + 1
ENDREAD
```

This example opens the IMAGE database PRODDB using a password of READER and an open mode of 5 and opens the database TESTDB using a password of WRITER and an open mode of 3. A variable named NUM is defined to count the records as they are read and is initialized to 1. The dataset MASTER-SET from the database PROD is read serially as long as counter NUM is less than or equal to 100. This limits the selection to 100 MASTER-SET records. The qualifying MASTER-SET records

are then copied to the MASTER-SET dataset in the test database. For each MASTER-SET record copied, all associated detail records from DETAIL-SET are read from the production database and copied to the test database. The counter variable NUM is incremented by 1.

**CREATE**

The CREATE statement is not supported for IMAGE databases.

**DELETE**

The DELETE statement may only be used on a database that has been opened for write access.

As required by IMAGE, when deleting from a master dataset, all associated detail records must have been deleted before the master record may be deleted.

DELETE may not be used against an automatic master dataset.

**Examples****Example 1**

```
OPEN DB IMAGE MYDB PASS=WRITER MODE=1
READ M = DB.MASTER-SET
    READ D = DB.DETAIL-SET &
        FOR KEY = M.KEY
        DELETE D
    ENDREAD
DELETE M
ENDREAD
```

This example opens the IMAGE database MYDB using a password of WRITER and an open mode of 1. The dataset MASTER-SET is read, then all corresponding detail records from DETAIL-SET are read and deleted. Only after the detail records have been deleted, are the master records deleted.

**LOCK**

The LOCK statement is used to issue locks on an IMAGE database. The LOCK statement can be used to issue locks at the database, dataset, or data item

level.

Before the LOCK statement can be applied to an IMAGE database, the locking level must be set to MANUAL.

**WARNING:** Before using the LOCK and UNLOCK statements, the user should fully understand database locking considerations. (See Appendix D of the *TurboIMAGE/XL Database Management System Reference Manual*.) Misuse of the LOCK statement can cause database deadlocks and data integrity problems.

#### Syntax

```
LOCK db-tag [lock-desc]
        [,lock-desc][,...]
```

db-tag specifies the database tag of the IMAGE database to which the LOCK statement is to apply.

lock-desc is one of the following:

```
BASE
ITEM set-name.item-name op expr
SET set-name
```

BASE indicates the that the entire database is to be locked. When BASE appears in a LOCK statement, it must appear by itself.

ITEM indicates the that the dataitem item-name within the dataset set-name is to be locked for the values indicated by op and expr. op must be either =, <=, or >=. expr may be any valid Warehouse expression.

SET indicates the that the dataset set-name is to be locked.

#### Examples

##### Example 1

```
OPEN ORD IMAGE ORDDb PASS=WRITER MODE=1
SET ORD LOCKING MANUAL
READ OM = ORD.ORDERS FOR STAT = "CANC"
LOCK ORD &
```

```

        ITEM ORD.LINES = OM.ORDNO, &
        ITEM ORD.COMMENTS = OM.ORDNO
    READ OL = ORD.LINES &
        FOR ORDNO = OM.ORDNO
        DELETE OL
    ENDREAD
    READ OC = ORD.COMMENTS &
        FOR ORDNO = OM.ORDNO
        DELETE OC
    ENDREAD
    UNLOCK ORD
ENDREAD

```

This example opens the IMAGE database ORDDb using a password of WRITER and an open mode of 1. Locking is set to MANUAL to enable use of the LOCK statement. The dataset ORDERS is read, selecting all orders with a status of CANC. A LOCK statement is applied that locks all records within the LINES dataset having an order number equal to ORDNO from the ORDERS dataset. The LOCK statement also locks all records within the COMMENTS dataset with order number ORDNO.

Records with an order number equal to the order number from ORDERS are then read and deleted from the datasets LINES and COMMENTS. After the records have been deleted, an UNLOCK statement unlocks the LINES and COMMENTS records.

## OPEN

The OPEN statement is used to access an IMAGE database.

### Syntax

```

OPEN db-tag IMAGE db-name
    [ PASS=db-password ]
    [ MODE=db-openmode ]
    [ EPASS1=encrypted-db-password ]
    [ CIU=ON ]
    [ DATA=IMAGE | IMAGE_ ]

```

db-tag is the database tag used to reference the database in the remainder of the script.

db-name is the file name of the database root file.

`db-password` is the database password used to access the database. If no database password is supplied, the default value of a semicolon (;) for database creator access is used.

`db-openmode` is the database mode used to open the database. If no `db-openmode` parameter is supplied, the default of 5 is used which allows read only access to the database. Acceptable values of `db-openmode` are:

- 1 Read/Write access.
- 2 Update, allow concurrent update.
- 3 Exclusive read/write access.
- 4 Read/Write, allow concurrent read.
- 5 Read only.
- 6 Read only, allow concurrent modify.
- 7 Exclusive read only access.
- 8 Read only, allow concurrent read.

`encrypted-db-password` is an encrypted database password for the database. An encrypted password for use in the `OPEN` statement may be generated by running Warehouse with `-c` (See **Checking Warehouse Server Connections** in [Chapter Seven](#)) or by DataBridger Studio. Password encryption is done by a proprietary algorithm based on the Data Encryption Standard (DES).

`CIU=ON` turns on critical item updates for the specified database. It has the same effect as the statement `SET db CIU ON`.

`IMAGE` accesses all datasets with `IMAGE` data types, which are big-endian (forward byte order). This is the default for MPE/iX Turbo `IMAGE` databases and HP-UX Eloquence databases.

`IMAGE_` accesses all datasets with `IMAGE_` data types, which are little-endian (reverse byte order). This is the default on Windows Eloquence databases and Linux Eloquence databases.

## Examples

Example 1

```
OPEN ODB IMAGE ORDERS.PUB.DB &
PASS=MYPASS MODE=1
```

This example opens the IMAGE database ORDERS.PUB.DB using a password of MYPASS and an open mode of 1. The database tag ODB is used to access this database in the remainder of the script.

Example 2

```
OPEN ORDERS IMAGE ORDERS PASS=READ
```

This example opens the IMAGE database ORDERS in the login group and account using a password of READ and the default open mode of 5. The database tag ORDERS is used to access this database in the remainder of the script.

## READ

The READ statement is used to read records from IMAGE datasets. Warehouse can read from manual master datasets, detail datasets, and automatic master datasets. Warehouse can also perform reverse serial and chained reads. If a third party indexing (TPI) product is installed and enabled, (e.g. Omnidex from DISC, or Superdex from Bradmark) Warehouse is able to take advantage of additional keys and partial keys in the READ statement.

## Syntax

```
READ read-tag = db-tag.dataset-name
    [ (RECNUM) ]
    [ (BACKWARDS) ]
    [ FORMAT format-name ]
    [ FOR condition ]
    [ ORDER BY order-list ]
```

When (RECNUM) is specified immediately after the dataset-name, Warehouse adds a virtual column named \$RECNUM to each IMAGE record. The record number may be used to read a specific record by referring to \$RECNUM. Record number access is activated only for the context of this READ

statement. To enable record number access for the entire database, see the Warehouse Statement `SET`.

When `(BACKWARDS)` is specified immediately after the `dataset-name` Warehouse performs a backwards read on the dataset. For serial reads, the entire dataset is read in reverse order. For chained reads, the chain is read in reverse order.

When a `FORMAT` is specified, `format-name` overrides the `IMAGE` definition of the dataset with the field names and types coming from `format` specified by `format-name` instead of from the `IMAGE` dataset definition.

When a `FOR` condition is specified, Warehouse attempts to optimize the `FOR` condition on the `READ` statement, reading by key item, or search item if possible.

When `ORDER BY` is specified, Warehouse first sorts the selected records, then reads the records in the order specified by `order-list`.

## Examples

### Example 1

```
OPEN CUST IMAGE CUSTDB.PUB.DB &  
  PASS=MYPASS MODE=3  
READ M = CUST.MASTER &  
  FOR STATUS = "CLOS" &  
  ORDER BY CUST-NAME  
READ D = CUST.ORDERS &  
  FOR CUST-NO = M.CUST-NO  
  PRINT CUST-NO, M.CUST-NAME, ORD-AMT  
ENDREAD  
ENDREAD
```

This example opens the `IMAGE` database `CUSTDB.PUB.DB` using a password of `MYPASS` and an open mode of 3. The database tag `CUST` is used to access this database in the remainder of the script. The master dataset `MASTER` is read and records where the `STATUS` field has a value of `CLOS` are selected. All qualifying records from `MASTER` are ordered by the value in the `CUST-NAME` field.

For each MASTER record, detail records from the ORDERS dataset are read *by key* using the search item CUST-NO. For each record from CUST.ORDERS, the fields CUST-NO and ORD-AMT are printed along with CUST-NAME from the CUST.MASTER dataset.

### Example 2

```
OPEN CUST IMAGE CUSTDB.PUB.DB &
    PASS=MYPASS MODE=3
READ M = CUST.MASTER &
    FOR STATUS = "CLOS" &
    ORDER BY CUST-NAME
    READ D = CUST.ORDERS (BACKWARDS) &
        FOR CUST-NO = M.CUST-NO
        PRINT CUST-NO, M.CUST-NAME, ORD-AMT
    ENDREAD
ENDREAD
```

This example is similar to the first example, except that the ORDERS dataset is read using a backwards chained read.

### Example 3

```
OPEN MYDB IMAGE ...
OPEN SRCDB ...
READ SRCORD = SRCDB.ORD
    READ ORD = MYDB.ORDS(RECNUM) &
        FOR $RECNUM = SRCORD.RECORD_NUMBER
        PRINT CUSTNO, ORDAMT
    ENDREAD
ENDREAD
```

Finds ORD from a source that contains the record number.

### Third Party Indexing (TPI)

Warehouse is capable of using third party indexes when executing a READ statements. If any data items have been indexed using a tool such as Omnidex from DISC or Superdex from Bradmark, Warehouse automatically uses those indexes when the indexed field is compared for equality (=). For indexed fields, wildcards and search expression may



be used. If a search expression is longer than the field type, the indexing tools require that the expression be terminated with a semicolon (;). To access a numeric TPI, use a string terminated by a semicolon.

Before Warehouse is able to take advantage of third party index, the database must have been enabled for indexing using the utility program supplied by third party index vendor. If the database has been enabled, the Warehouse `SHOW db-tag` command will show that TPI is `ON` for the database. If the `SHOW` command does not show `ON` for the database, Warehouse will not be able to take advantage of third party indexes.

For more information on setting up a third party indexes and for search expressions, see the documentation for your third party indexing product.

#### TPI Example

```
OPEN CDB IMAGE CUSTDB PASS=MYPASS MODE=1
READ M = CDB.CUSTMAST FOR ZIP-CODE = "9@" ;
    PRINT CUST-NO,
    PRINT CUST-NAME,
    PRINT CUST-CITY,
    PRINT CUST-STATE,
    PRINT ZIP-CODE
ENDREAD
```

This example opens the IMAGE database CUSTDB using a password of MYPASS and an open mode of 1. The database tag CDB is used to access this database in the remainder of the script. The dataset CUSTMAST is read and all records that have a ZIP-CODE field starting with 9 are selected. This is possible because a third party index has been created for the ZIP-CODE field.

#### `$DATASETS`

A virtual dataset called `$DATASETS` is available to IMAGE databases. `$DATASETS` is read-only and contains a list of datasets in the database and their

attributes. The fields in \$DATASETS are:

SETNAME	X16	Name of the IMAGE data set
SETTYPE	X2	Dataset type: A, D, or M
ENTRYLEN	I1	Number of double-bytes in each record
BLOCKFACT	I1	Blocking factor of the records
NUMENTRIES	I2	Number of records in the dataset
CAPACITY	I2	Maximum number records dataset can hold

#### \$DATASETS Example

```
OPEN CDB IMAGE CUSTDB PASS=MYPASS MODE=1
READ M = CDB.$DATASETS FOR SETTYPE <> "A"
  PRINT SETNAME,
  PRINT SETTYPE,
  PRINT NUMENTRIES,
  PRINT CAPACITY
ENDREAD
```

This example opens the IMAGE database CUSTDB using a password of MYPASS and an open mode of 1 and a database tag CDB. The virtual dataset \$DATASETS is read selecting all records except automatic masters.

## SET

The SET statement is used to set IMAGE access options.

### Syntax

```
SET db-tag image-option value
```

db-tag specifies the database tag of the IMAGE database to which the SET statement is to apply.

image-option specifies the name of the option to be changed. image-option must be one of the following for IMAGE databases:

CIU	Sets critical item update
DEFER	Sets defer mode
LOCKING	Sets locking option
MAXOPENS	Sets maximum number of internal DBOPENS

RECNUMS	Enables record number access for the entire database.
TPI	Sets third party indexing

value is the new value of the option as follows:

CIU	value must be either ON to enable critical item update, or OFF to disable critical item update. To turn critical item update on, critical item update must be allowed by the database administrator using DBUTIL.PUB.SYS.
-----	---

DEFER	value must be either ON to enable write defer mode, or OFF to disable write defer mode. When write defer mode is on, performance of database writes and deletes is much better because IMAGE buffers are only written to disk as necessary. DEFER may only be used when the database has been opened in mode 3, exclusive access.
-------	---

**WARNING:** If a system failure occurs while using an IMAGE database in write defer mode, the database can easily be corrupted.

LOCKING	value must be either BASE, MANUAL, OFF, ROLLBACK, or SET.
---------	---

BASE Indicates that Warehouse is to lock at the *database* level. Each time a dataset is accessed, the database is locked until the

ENDREAD for the outermost READ statement is reached.

**MANUAL** Indicates that only the LOCK and UNLOCK statements are to be used for database locking. SET LOCKING MANUAL must be issued on the database before Warehouse will accept LOCK and UNLOCK statements.

When LOCKING is set to MANUAL the user is responsible for inserting LOCK and UNLOCK statements at the appropriate points in the script file.

**OFF** Indicates that Warehouse is to do no database locking.

**ROLLBACK** Indicates that Warehouse is to use database locking and call the Image intrinsics DBXBEGIN, DBXEND, and DBXUNDO for transaction control.

**SET** Indicates that Warehouse is to lock at the *dataset* level. Each time a dataset is accessed, the dataset is locked until the ENDREAD for the

outermost READ  
statement is reached.

MAXOPENS	To access the same dataset more than once concurrently, Warehouse may open the database multiple times. Setting MAXOPENS controls the maximum number of times that Warehouse may open the database. Setting MAXOPENS to 1 limits the number of database opens to one. The default value of MAXOPENS is 3 for database open modes of 1 and 5 and the default value is 1 for all other database open modes.
RECNUM	value must be either ON or OFF. If enabled, a virtual column named \$RECNUM is appended to each IMAGE record. In addition, the record number may be used to read a specified record by referring to a value in \$RECNUM.
TPI	value must be either ON to enable use of third party indexes or OFF to disable use of third party indexes. This option is maintained by Warehouse and does not generally need to be changed. It is provided to disable the use of third party indexes in the unusual case where a database has third party indexes, but you do not wish Warehouse to use them.

Examples

Example 1

```
OPEN C IMAGE CUST PASS=THEPASS MODE=3  
SET C DEFER ON
```

This example opens the IMAGE database CUST using a password of THEPASS and an open mode of 3. The database tag C is used to access the database in the remainder of the script. Write defer mode is set ON, improving the performance of database writes and deletes. Defer mode is only allowed because the database was opened in mode 3, exclusive access.

### Example 2

```
OPEN ORDDB IMAGE ORDDB.DATABASE PASS=IO  
SET ORDDB LOCKING OFF
```

This example opens the IMAGE database ORDDB.DATABASE using a password of IO and the default open mode of 5. The database tag ORDDB is used to access the database in the remainder of the script. The open mode of 5 causes the initial locking mode to be SET. The SET statement changes the locking to OFF so that no locking takes place when the database is accessed.

## UNLOCK

The UNLOCK statement is used to release locks on an IMAGE database set by the LOCK statement. The UNLOCK statement release all locks on the specified database.

Before the UNLOCK statement can be applied to an IMAGE database, the locking level must be set to MANUAL.

**WARNING:** Before using the LOCK and UNLOCK statements, the user should fully understand database locking considerations. (See Appendix D of the *TurboIMAGE/XL Database Management System Reference Manual*: <http://docs.hp.com/cgi-bin/doc3k/B3039190010.17091/28>) Misuse of the UNLOCK statement can cause database deadlocks

and data integrity problems.

**Syntax**

UNLOCK db-tag

db-tag specifies the database tag of the IMAGE database to which the UNLOCK statement is to apply.

**Examples****Example 1**

```
OPEN ORD IMAGE ORDDB PASS=WRITER MODE=1
SET ORD LOCKING MANUAL
LOCK ORD SET ORDERS
READ OM = ORD.ORDERS FOR STAT = "CANC"
  DELETE OM
ENDREAD
UNLOCK ORD
```

This example opens the IMAGE database ORDDB using a password of WRITER and an open mode of 1. The LOCK statement is used to lock the entire ORDERS dataset. ORDERS is read, selecting all orders with a status of CANC, and all selected orders are deleted. After all processing on the ORDERS dataset is complete, an UNLOCK statement releases the lock on the ORDERS dataset.

**UPDATE**

The UPDATE statement is used to update a field within the current record of a READ statement. The UPDATE statement may only be used on a database that has been opened for update access.

Critical items (key fields, search items, sort items) may only be updated if critical item update has been enabled for the database.

UPDATE may not be done against an automatic master dataset.

**Examples****Example 1**

```
OPEN C IMAGE CUST PASS=THEPASS MODE=1
CREATE A ARCHIVE ARFILE
```

```
READ M = C.MAST FOR DATE <= 921231
  COPY M TO A.MAST
  UPDATE M SET STATUS = "ARCH"
ENDREAD
```

This example opens the IMAGE database CUST using a password of THEPASS and an open mode of 1. The database tag C is used to access this database in the remainder of the script. An archive file called ARFILE is created, and the database tag A is used to access the file. The master dataset MAST is read *serially* and records where the DATE field has a value less than or equal to 921231 are selected. All qualifying records from MAST are copied to the archive file, and the field STATUS is updated to the value of ARCH.



## ODBC

## ODBC file access

This section describes the considerations for each Warehouse statement that accesses an ODBC database.

ODBC (Open Database Connectivity) is a database interface standard from the Microsoft Corporation. ODBC is an interface standard that can be adapted to practically any type of commercial database. To access a database using ODBC, an ODBC driver must exist on your machine and is typically provided by your operating system or database vendor.

Warehouse supports Level 2 compliant ODBC connections on the following database / operating system platforms:

Microsoft SQL Server on Microsoft Windows

Prior to accessing a database through ODBC, an ODBC data source must be set up using the ODBC32 or ODBC64 control panel.

## COPY...TO

Records may only be copied to an ODBC table or updateable view.

Read-only columns are not copied when copying to an ODBC database. This provides support for SQL Server identity columns. When an attempt is made to copy to a table containing an identity column, Warehouse does not attempt to write to the identity column.

## Syntax

```
COPY record TO output-table
    [ (IDENTITY_INSERT) ]
    [ (HINT locking-hint) ]
```

`record` is the name of a record created with either the `DEFINE` statement or a read tag created by the `READ` statement.

`output-table` is the name of the ODBC table to which the record is copied.

`identity-insert` makes Warehouse write all columns to the target table, including any identity columns. Without the `IDENTITY_INSERT` option, identity columns are not written to the target table.

`locking-hint` is used to specify a SQL Server 7 locking hint when copying to a SQL Server 7 datasource. See the SQL Server 7 documentation for information on the effect of the locking hints at [http://msdn.microsoft.com/en-us/library/aa213026\(SQL.80\).aspx](http://msdn.microsoft.com/en-us/library/aa213026(SQL.80).aspx). Locking hints available are:

<code>HOLDLOCK</code>	<code>NOLOCK</code>
<code>PAGLOCK</code>	<code>READCOMMITTED</code>
<code>READPAST</code>	<code>READUNCOMMITTED</code>
<code>REPEATABLEREAD</code>	<code>ROWLOCK</code>
<code>SERIALIZABLE</code>	<code>TABLOCK</code>
<code>TABLOCKX</code>	<code>UPDLOCK</code>

## Examples

### Example 1

```
OPEN ARCH ARCHIVE ARCFIL
OPEN PROD ODBC ORDERS
READ ORDS = ARCH.ORDS FOR DATE = 920222
  COPY ORDS TO PROD.ORDDB.ORDS
  READ LINES = ARCH.ORD_LINES &
    FOR ORDNO = ORDS.ORDNO
    COPY LINES TO PROD.ORDDB.ORDLINES
  ENDREAD
ENDREAD
```

This example opens the archive file `ARCFIL`, and opens the ODBC database `ORDERS`. All `ORDS` records with a `DATE` field equal to 920222 are read from the archive file. The `ORDS` records are then copied into the table `ORDDB.ORDS` in the ODBC database. The associated `ORD_LINES` records are then read from the archive file and copied into the table `ORDDB.ORDLINES`.

### Example 2

```
OPEN SRCDB REMOTE ...
```

```

OPEN TGTDB ODBC MYODBCDB

DEFINE TGTORDREC : USING TGTDB.ORDERS

READ SRCCUST_R = SRCDB.ORDERS
// CUSTOMERID is an identity column
SETVAR TGTORDREC.CUSTOMERID = &
    SRCCUST_R.CUSTOMERID
SETVAR TGTORDREC.NAME = SRCCUST_R.NAME
SETVAR TGTORDREC.STATUS = &
    SRCCUST_R.STATUS
SETVAR TGTORDREC.TOTAL = &
    SRCCUST_R.TOTAL
SETVAR TGTORDREC.LINEITEMCOUNT = &
    SRCCUST_R.LINEITEMCOUNT

COPY TGTORDREC TO &
    TGTDB.ORDERS( IDENTITY_INSERT)
ENDREAD

```

This example copies `ORDERS` from the source to the target where `CUSTOMERID` is a SQL Server identity column. The `( IDENTITY_INSERT)` causes the `CUSTOMERID` value in the target to be same as in the source. Without `( IDENTITY_INSERT)`, the `CUSTOMERID` would be assigned automatically by SQL Server and would probably not contain the same value as the source record.

CREATE

The `CREATE` statement is not supported for ODBC databases.

DELETE

The `DELETE` statement may only be used on a table or updateable view.

### Examples

#### Example 1

```

OPEN DB ODBC MYDB USER=MYUSER PASS=MYPS
READ M = DB.PARTS.MASTER &
    FOR STATUS = "NOTREF"
    DELETE M
ENDREAD

```

This example opens the ODBC database `MYDB`

using a user name of MYUSER and a user password of MYPS. The table PARTS.MASTER is read for all parts that have NOTREF in the STATUS field. All selected records are then deleted.

## OPEN

The OPEN statement is used to access an ODBC database.

### Syntax

```
OPEN db-tag ODBC data-source  
    [USER=user-name]  
    [PASSWORD=user-password]  
    [EPASS1=encrypted-password]  
    [SCHEMA=schema-name]
```

db-tag is the database tag used to reference the database in the remainder of the script.

data-source is the name of the ODBC data source as it has been set up using the ODBC32 Control Panel. If the data source name has spaces or special characters, it must be enclosed in double quotes.

user-name is the name of the user accessing the data source. If no user-name is specified, a trusted connection is used.

password is the password for user-name.

encrypted-password is an encrypted password for user-name on the ODBC data source. An encrypted password for use in the OPEN statement may be generated by running Warehouse with -c (See **Checking Warehouse Server Connections** in [Chapter Seven](#)) or by DataBridger Studio. Password encryption is done by a proprietary algorithm based on the Data Encryption Standard (DES).

Schema-name sets the default schema for the database.

### Examples

[Example 1](#)

```
OPEN ORD ODBC MYODBCDB &
  USER=MGR PASSWORD=EAGLE
```

Warehouse is run on Windows NT and an ODBC data source named MYODBCDB is opened using a user name MGR a password EAGLE. The database is assigned a Warehouse database tag of ORD to reference the database in the remainder of the script.

## READ

The READ statement is used to read records from ODBC tables and views.

## Syntax

```
READ read-tag = db-tag.table-ref
  [ ( HINT locking-hint ) ]
  [FOR condition]
  [ORDER BY order-list]
```

table-ref is the name of the table or view Warehouse is to read. table-ref is one of:

```
owner.table-name
owner.view-name
table-name
view-name
```

locking-hint is used to specify a SQL Server 7 locking hint when reading from a SQL Server 7 datasource. See the SQL Server 7 documentation for information on the effect of the locking hints at [http://msdn.microsoft.com/en-us/library/aa213026\(SQL.80\).aspx](http://msdn.microsoft.com/en-us/library/aa213026(SQL.80).aspx)  
Locking hints available are:

HOLDLOCK	NOLOCK
PAGLOCK	READCOMMITTED
READPAST	READUNCOMMITTED
REPEATABLEREAD	ROWLOCK
SERIALIZABLE	TABLOCK
TABLOCKX	UPDLOCK

When a FOR condition is specified, Warehouse selects only those records matching condition. If a FOR condition is not specified, all records in the table or view are selected.

When ORDER BY is specified, Warehouse orders (sorts) the records as specified by order-list.

### Examples

#### Example 1

```
OPEN SALES ODBC SALEINFO
READ M = SALES.CUST.MASTER &
        (HINT NOLOCK) &
        FOR STATUS = "CLOS" &
        ORDER BY CUST_NAME
READ D = SALES.CUST.ORDERS &
        FOR CUST_NO = M.CUST_NO
        PRINT CUST_NO, M.CUST_NAME, ORD_AMT
ENDREAD
ENDREAD
```

This example opens the ODBC data source called SALEINFO. The database tag SALES is used to access this database environment in the remainder of the script. The customer master table CUST.MASTER is read using the SQL Server locking hint NOLOCK and records where the STATUS field has a value of CLOS are selected. All qualifying records from CUST.MASTER are ordered by the value in the CUST\_NAME field. For each CUST.MASTER record, detail records from the CUST.ORDERS table are read using the indexed field CUST\_NO. For each record from CUST.ORDERS, the fields, CUST\_NO and ORD\_AMT are printed along with CUST\_NAME from the CUST.MASTER table.

### SET

The SET statement is used to set ODBC access options.

### Syntax

```
SET db-tag odbc-option value
```

db-tag specifies the database tag of the ODBC database to which the SET statement is to apply.

odbc-option specifies the name of the option to be changed. odbc-option must be one of the following for ODBC databases:

AUTOCOMMIT	Value of ON enables the ODBC autocommit feature. Value of OFF disables the ODBC autocommit feature. Under usual situations Warehouse handles database commits and setting AUTOCOMMIT is not necessary. This feature is provided for unusual data sources and situations.
LONGSIZE	Sets the maximum length of the LONG data types. When reading from or writing to an ODBC database, LONGSIZE controls the maximum number of characters (bytes) that a LONG VARBINARY or LONG VARCHAR field may contain. The default value of LONGSIZE is 10,000.
MAXHANDLES	Sets the maximum number of ODBC handles that Warehouse may have open at a time. While Warehouse is running, ODBC handles are opened and closed as necessary. The default value of MAXHANDLES is 255. Each ODBC handle utilizes system resources and it may be desirable to user fewer handles.
ODBCTRACE	Value must be either OFF to disable ODBC tracing, or a filename containing the name of the ODBC trace file. When tracing is on, each ODBC call is logged to the trace file by the ODBC

driver. Enabling ODBC tracing has a significant impact on performance should only be used to debug rare situations.

## SHOWSQL

Value of ON enables display of the SQL statements Warehouse uses for database access which can sometimes be useful in debugging scripts. A value of OFF, the default, disables display of SQL statements.

## TRANS

Value must be either OFF to disable database transactions, or ON to enable database transactions. When transactions are disabled, Warehouse does not do commits and rollbacks against the database. The default value of TRANS is ON for all datasources.

## Examples

Example 1

```
OPEN C ODBC XDB
SET C LONGSIZE 250000
SET C MAXHANDLES 100
```

This example opens an ODBC data source called XDB. The database tag C is used to access the database in the remainder of the script. The LONGSIZE is set to 250,000 allowing up to 250,000 characters to be read into or written from LONG and LONG RAW fields. The maximum number of handles is set to 100 to minimize system resources.

## SHOW

The SHOW statement is used to display ODBC attributes about an ODBC data source.



**Syntax**                    `SHOW db-tag [odbc-option] [table-name]`

`db-tag` specifies the database tag of the ODBC database for which attributes are to be displayed.

`odbc-option` specifies the name of the option to be changed. If no `odbc-option` is specified, the Warehouse options for the database are displayed.

ALL	Indicates that Warehouse options, along with DRIVER, CONNECT and TYPES (see below) information is displayed.
CONNECT	Displays the ODBC connection options. The meaning of the options displayed in documented in the ODBC manual.
DRIVER	Displays ODBC driver information.
KEYS	Displays key information for <code>table-name</code> . The <code>table-name</code> parameter is required when using KEYS.
TABLE	Displays information about each field in <code>table-name</code> . The <code>table-name</code> parameter is required when using TABLE.
TYPES	Displays information about the data types supported by this ODBC data source.

**Examples****Example 1**

```
1> OPEN DB ODBC MyPTA
2> SHOW DB
SQL Server ODBC Data Source
File name      : mypta
Long size     : 10000
```

```
Auto commit : OFF
MAXHANDLES  : 255
TRANS       : ON
Case        : ON
Quote       : ""
SETPOS      : OFF

3> SHOW DB CALLIST
RECORD // 194 bytes
  ID      : ODBC INTEGER OFFSET 1
  NAME    : ODBC CHAR(25) ALLOW NULLS
           OFFSET 9
  ADDR    : ODBC VARCHAR(50) ALLOW NULLS
           OFFSET 41
  PHONE   : ODBC DECIMAL(15,0) ALLOW
           NULLS OFFSET 97
  Email   : ODBC VARCHAR(50) ALLOW NULLS
           OFFSET 121
  TXNDATE : ODBC TIMESTAMP ALLOW NULLS
           OFFSET 177
END
```

This example opens an ODBC data source called MyPTA that is tied to an SQL Server database. The database tag DB is used to access the database in the remainder of the script. Information about the ODBC connection is shown in line 2. Information about table CALLIST is shown in line 3.

### Example 2

```
1> OPEN DB ODBC UPSELL
2> SHOW DB
MS Access
  ODBC Data Source
File name   : upsell
Long size   : 10000
Auto commit : ON
MAXHANDLES  : 255
TRANS       : ON
Case        : ON
Quote       : "`"
MSACCESS    : ON
SETPOS      : OFF

3> SHOW DB TDWDS
RECORD // 696 bytes
  DSNAME      : ODBC VARCHAR(20) ALLOW
```

```

                                NULLS OFFSET 1
DSDATE                        : ODBC VARCHAR(20) ALLOW
                                NULLS OFFSET 25
DSTYPE                        : ODBC VARCHAR(20) ALLOW
                                NULLS OFFSET 49
DSCONNECTSTR                  : ODBC VARCHAR(255)
                                ALLOW NULLS OFFSET 73
DSCOMMENT                     : ODBC VARCHAR(20) ALLOW
                                NULLS OFFSET 337
DSOPEN                        : ODBC SMALLINT ALLOW
                                NULLS OFFSET 361
USERPW                        : ODBC VARCHAR(144)
                                ALLOW NULLS OFFSET 369
DBPW                          : ODBC VARCHAR(144)
                                ALLOW NULLS OFFSET 521
CFGNAME                       : ODBC VARCHAR(20) ALLOW
                                NULLS OFFSET 673
END

```

This example mimics Example #1 but with an MS Access database.

### Example #3

```

1> open db ODBC MyContacts
2> show db
MySQL ODBC Data Source
File name      : MyContacts
Long size     : 10000
Auto commit   : ON
MAXHANDLES    : 255
TRANS         : ON
Case          : OFF
Quote         : "`"
SETPOS        : OFF
FORUPDATE     : ON

3> show db CONTACTS
RECORD // 194 bytes
ID       : ODBC SMALLINT ALLOW NULLS
          OFFSET 1
NAME     : ODBC VARCHAR(25) ALLOW
          NULLS OFFSET 9
ADDR     : ODBC VARCHAR(50) ALLOW
          NULLS OFFSET 41
PHONE    : ODBC DECIMAL(15,0) ALLOW
          NULLS OFFSET 97
EMAIL    : ODBC VARCHAR(50) ALLOW
          NULLS OFFSET 121
TXNDATE  : ODBC TIMESTAMP ALLOW NULLS

```

```

                                OFFSET 177
END

4> SHOW DB CONNECT
Connect Options
-----
SQL_ACCESS_MODE = SQL_MODE_READ_WRITE
SQL_AUTOCOMMIT = SQL_AUTOCOMMIT_ON
SQL_LOGIN_TIMEOUT(secs) = 0
SQL_ODBC_CURSORS = SQL_CUR_USE_DRIVER
SQL_OPT_TRACE = SQL_OPT_TRACE_OFF
SQL_CURRENT_QUALIFIER = "testdb"
SQL_CURSOR_COMMIT_BEHAVIOR =
SQL_CB_PRESERVE
SQL_POSITIONED_STATEMENTS:
    SQL_PS_POSITIONED_DELETE = YES
    SQL_PS_POSITIONED_UPDATE = YES
    SQL_PS_SELECT_FOR_UPDATE = NO
SQL_ACTIVE_CONNECTIONS =
SQL_ACTIVE_STATEMENTS =

5> SHOW DB DRIVER

SQL_DBMS_NAME = MySQL
SQL_DRIVER_NAME = myodbc3.dll
Driver conformance level = Level 1
SQL conformance level = Core

6> SHOW DB TYPES

Type Name  WH Type      Num    LocTyp
-----
bit        ODBC BIT      -7     bit(1)
tinyint    ODBC TINYINT  -6     tinyint
. . .

```

This example mimics Example #1 but with a MySQL database.

## UPDATE

The UPDATE statement may only be used on a table or updateable view.

## Examples

### Example 1

```

OPEN C ODBC MFIL USER=JONES PASS=AAA
CREATE A ARCHIVE ARFILE
READ M = C.DB.MAST FOR STATUS = "CLOS"
COPY M TO A.DB.MAST

```

```
UPDATE M SET STATUS = "ARCH"  
ENDREAD
```

This example opens an ODBC data source called `MFIL` using a user name of `JONES` and a user password of `AAA`. The database tag `C` is used to access the database in the remainder of the script. An archive file called `ARFILE` is created using the database tag `A`. The table `DB.MAST` is read and records where the `STATUS` field has a value equal to `CLOS` are selected. All qualifying records from `DB.MAST` are copied to the archive file and the field `STATUS` is updated to the value of `ARCH`.

## ORACLE

## Oracle file access

This section describes the considerations for each Warehouse statement that accesses an Oracle database.

Oracle is a database from the Oracle Corporation that runs on many different operating systems including all supported Warehouse platforms.

## COPY...TO

Records may only be copied to an Oracle table or updateable view. It is possible to assign the value of a column to an Oracle sequence.

## Syntax

```
COPY record TO output-table
      [(SEQ column = sequence.NEXTVAL)]
```

`record` is the name of a record created with either the `DEFINE` statement or a read tag created by the `READ` statement.

`output-table` is the name of the Oracle table to which the record is copied.

`column` is the name of a column within the `output-table`. For each record copied to the table `column` is assigned the next value in the Oracle sequence specified by `sequence`. This allows a column to automatically be assigned a unique value by Oracle.

`sequence` is the name of an Oracle sequence that has been created by the database administrator. Note that `sequence` must be following by `.NEXTVAL`.

## Examples

Example 1

```
OPEN ARCH ARCHIVE ARCFIL
OPEN PROD ORACLE SCOTT/TIGER
READ ORDS = ARCH.ORDS FOR DATE = 920222
  COPY ORDS TO PROD.ORDDB.ORDS
```

```

      READ LINES = ARCH.ORD_LINES &
        FOR ORDNO = ORDS.ORDNO
      COPY LINES TO PROD.ORDDB.ORDLINES
    ENDREAD
  ENDREAD

```

This example opens the archive file ARCFIL, and opens the Oracle database with a user name of SCOTT and a user password of TIGER. All ORDS records with a DATE field equal to 920222 are read from the archive file. The ORDS records are then copied into the table ORDDB.ORDS in the Oracle database. The associated ORD\_LINES records are then read from the archive file and copied into the table ORDDB.ORDLINES.

### Example 2

```

OPEN ARCH ARCHIVE ARCFIL
OPEN PROD ORACLE SCOTT/TIGER
READ ORDS = ARCH.ORDS FOR DATE = 920222
  COPY ORDS TO PROD.ORDS &
    (SEQ ORDNO = ORDSEQ.NEXTVAL)
  ENDREAD

```

This example opens the archive file ARCFIL, and opens the Oracle database with a user name of SCOTT and a user password of TIGER. All ORDS records with a DATE field equal to 920222 are read from the archive file. The ORDS records are then copied into the table ORDS in the Oracle database. As each record is copied to ORDS the column ORDNO is assigned a the next value in the Oracle sequence ORDSEQ.

### APPEND

Using the append hint adds the APPEND hint to the Oracle INSERT statement used to copy the data and may increase the performance when writing to an Oracle table. See the Oracle SQL Reference Manual for more information.

### Syntax

```

COPY rec TO db-tag.target-table (HINT
APPEND)

```

**CREATE** The **CREATE** statement is not supported for Oracle databases.

**DELETE** The **DELETE** statement may only be used on a table or updateable view.

**Examples** Example 1

```
OPEN DB ORACLE USER/PASS
READ M = DB.PARTS.MASTER &
      FOR STATUS = "NOTREF"
      DELETE M
ENDREAD
```

This example opens the Oracle database using a user name of `USER` and a user password of `PASS`. The table `PARTS.MASTER` is read for all parts that have `NOTREF` in the `STATUS` field. All selected records are then deleted.

**OPEN** The **OPEN** statement is used to access an Oracle database.

**Note:** Only one Oracle environment may be opened within a Warehouse script. If you wish to open more than one Oracle instance, you may use a `REMOTE` database to open other Oracle instances.

**Syntax**

```
OPEN db-tag ORACLE user-name/password
      [EPASS1=encrypted-password]
      [HOME=oracle-home]
      [SID=oracle-sid]
```

`db-tag` is the database tag used to reference the database in the remainder of the script.

`user-name/password` is the user name and password that is used to access the Oracle instance. It may be any valid Oracle connect string.

`encrypted-password` is an encrypted password for `user-name` on the Oracle database. An



encrypted password for use in the `OPEN` statement may be generated by running Warehouse with `-c` (See **Checking Warehouse Server Connections** in [Chapter Seven](#)) or by DataBridger Studio. Password encryption is done by a proprietary algorithm based on the Data Encryption Standard (DES).

`oracle-home` is the home directory of the Oracle instance. (On MPE/iX, this directory must be in HFS syntax, e.g. `/ACCT/GROUP`.) If `oracle-home` is not specified, Warehouse uses the `ORACLE_HOME` environment variable.

`oracle-sid` is the system ID of the Oracle instance. If `oracle-sid` is not specified, Warehouse uses the `ORACLE_SID` environment variable.

### Examples

#### Example 1

```
OPEN ORD ORACLE SCOTT/TIGER &  
HOME=/ORATAUR/PUB SID=A
```

Warehouse is run on MPE/iX and an Oracle database is opened using a user name SCOTT, a password TIGER, an Oracle home of `/ORATAUR/PUB`, and an Oracle system ID of A. The database is assigned a Warehouse database tag of ORD to reference the database in the remainder of the script.

### READ

The `READ` statement is used to read records from Oracle tables and views.

### Syntax

```
READ read-tag = db-tag.table-ref  
    [FOR condition]  
    [ORDER BY order-list]
```

`table-ref` is the name of the table or view Warehouse is to read. `table-ref` is one of:

```
owner.table-name  
owner.view-name  
table-name
```

view-name

When a FOR condition is specified, Warehouse selects only those records matching condition. If no FOR condition is specified, all records in the table or view are selected.

When ORDER BY is specified, Warehouse orders (sorts) the records as specified by order-list.

## Examples

### Example 1

```
OPEN SALES ORACLE SCOTT/TIGER
READ M = SALES.CUST.MASTER &
        FOR STATUS = "CLOS" &
        ORDER BY CUST_NAME
  READ D = SALES.CUST.ORDERS &
        FOR CUST_NO = M.CUST_NO
    PRINT CUST_NO, M.CUST_NAME, ORD_AMT
  ENDREAD
ENDREAD
```

This example opens the Oracle database using a user name of SCOTT and a user password of TIGER. The database tag SALES is used to access this database environment in the remainder of the script. The customer master table CUST.MASTER is read and records where the STATUS field has a value of CLOS are selected. All qualifying records from CUST.MASTER are ordered by the value in the CUST\_NAME field. For each CUST.MASTER record, detail records from the CUST.ORDERS table are read using the indexed field CUST\_NO. For each record from CUST.ORDERS, the fields, CUST\_NO and ORD\_AMT are printed along with CUST\_NAME from the CUST.MASTER table.

## SET

The SET statement is used to set Oracle access options.

## Syntax

```
SET db-tag oracle-option value
```

db-tag specifies the database tag of the Oracle database to which the SET statement is to apply.

`oracle-option` specifies the name of the option to be changed. `oracle-option` must be one of the following for Oracle databases:

BULK	Setting BULK causes Warehouse to internally buffer inserts to Oracle tables and to periodically send them to Oracle as a group. This can increase performance when inserting many records to a single table. The value of BULK should be a number less than your commit rate that indicates how many records to group before sending them to Oracle. Note: The commit rate should be a multiple of the BULK rate so that bulked records are committed properly. e.g. COMMITRATE of 1000, and BULK of 200.
LONGSIZE	Sets the maximum length of the LONG and LONG RAW Oracle data types. When reading from or writing to an Oracle database, LONGSIZE controls the maximum number of characters (bytes) that a LONG or LONG RAW field may contain. The default value of LONGSIZE is 10,000.
SHOWSQL	Value of ON enables display of the SQL statements Warehouse uses for database access which can sometimes be useful in debugging scripts. A value of OFF, the

default, disables display of SQL statements.

### Examples

#### Example 1

```
OPEN C ORACLE SYSTEM/MANAGER
SET C LONGSIZE 250000
```

This example opens an Oracle database using a user name of SYSTEM and a user password of MANAGER. The database tag C is used to access the database in the remainder of the script. The LONGSIZE is set to 250,000 allowing up to 250,000 characters to be read into or written from LONG and LONG RAW fields.

### UPDATE

The UPDATE statement may only be used on a table or updateable view.

### Examples

#### Example 1

```
OPEN C ORACLE JONES/AAA
CREATE A ARCHIVE ARFILE
READ M = C.DB.MAST FOR STATUS = "CLOS"
    COPY M TO A.DB.MAST
    UPDATE M SET STATUS = "ARCH"
ENDREAD
```

This example opens an Oracle database using a user name of JONES and a user password of AAA. The database tag C is used to access the database in the remainder of the script. An archive file called ARFILE is created using the database tag A. The table DB.MAST is read and records where the STATUS field has a value equal to CLOS are selected. All qualifying records from DB.MAST are copied to the archive file, and the field STATUS is updated to the value of ARCH.

**REMOTE****REMOTE database and file access**

This section describes the considerations for each Warehouse statement that accesses a remote database or file.

Accessing a REMOTE file or database causes a network TCP/IP connection to be made to a Warehouse server program running on the remote system. After the connection is made to the remote system, the Warehouse client sends data and messages back and forth to the Warehouse server. This allows a remote file or database to behave exactly like a local database of the same type.

In order for a REMOTE connection to be established with the OPEN or CREATE statement, a Warehouse server program must be running on the remote system and the remote system must have an AUTHFILE entry that permits access. For more information on the AUTHFILE and setting up the Warehouse server see [Chapter Seven, Installation and Execution](#).

**COPY...TO**

The COPY statement copies a record across the network to the remote database or file. The restrictions for COPY are the same as those for the remote file or database type.

**Syntax**

```
COPY record TO output-file  
    [FORMAT format-name]  
    [[;] ERRORS TO error-file]  
    [[;] WAIT | NOWAIT]
```

`record` is the name of a record created with either the DEFINE statement or a read tag created by the READ statement.

`output-file` is the name of the remote file to which the record is copied. When copying to a database, `output-file` is in the format `db-tag.table-name`.

`format-name` is the name of a format previously

created with the `FORMAT` statement. When `format-name` is specified, the format of the `output-file` is redefined to be that of the format specified by `format-name`.

`error-file` is the name of the file to which the record is copied in the event of an error while copying to `output-file`. When copying errors to a database, `error-file` is in the format `db-tag.table-name`.

When `ERRORS TO` is specified the Warehouse client keeps an internal local copy of each record copied to the remote database. When the Warehouse server successfully writes the record to the database, a success message is sent to the Warehouse client and the local copy on the client is deleted. If the server gets an error writing the record, an error message is sent to the client and the client then writes the local copy of the record to `error-file`.

`NOWAIT` causes Warehouse to continue processing the script without waiting for the Warehouse server to actually copy the record to the remote database. This allows Warehouse to continue processing the script without waiting for a response from the server. `NOWAIT` is default when copying to a remote database.

**Warning:** Using `NOWAIT` (the default) can have error recovery implications when used inside a `TRY/RECOVER` block. When error occurs on a `NOWAIT COPY`, the `RECOVER` statement will NOT be entered with the record that had the error.

`WAIT` only has an effect when copying to a remote database. `WAIT` causes Warehouse to wait for the server to actually copy the record to the database before continuing. `WAIT` is typically used to enhance error recovery within a `TRY` block.

Using `WAIT` can have significant performance implications since the Warehouse client must wait for a response from the server for each record

written.

### Examples

#### Example 1

```
OPEN ARCH ARCHIVE ARCFIL
OPEN PROD REMOTE BLUE &
  USER=BLUEUSER PASSWORD=MYPASS &
  ORACLE SCOTT/TIGER
READ ORDS = ARCH.ORDS FOR DATE = 920222
  COPY ORDS TO PROD.ORDDDB.ORDS
  READ LINES = ARCH.ORD-LINES &
    FOR ORDNO = ORDS.ORDNO
    COPY LINES TO PROD.ORDDDB.ORDLINES
ENDREAD
ENDREAD
```

This example opens the local archive file ARCFIL and opens a remote Oracle database on the system named BLUE. A user name of BLUEUSER and a password of MYPASS are used to logon to the remote system. The remote Oracle database on the system BLUE is opened by the Warehouse server using an Oracle user name of SCOTT and an Oracle password of TIGER. All ORDS records with a DATE field equal to 920222 are read from the archive file. The ORDS records are then copied across the network into the table ORDDDB.ORDS in the Oracle database on the system BLUE. The associated ORD-LINES records are then read from the archive file and copied across the network into the table ORDDDB.ORDLINES.

#### Example 2

```
OPEN IMG IMAGE PRODDDB PASS=MP MODE=5
OPEN ORA REMOTE BLUE &
  USER=ORAUSER PASSWORD=ORAPASS &
  ORACLE SCOTT/TIGER
READ ORD = IMG.ORD FOR STAT = "CLOS"
  TRY
    COPY ORD TO PROD.ORDDDB.ORDS ; WAIT
  RECOVER
    PRINT ORDNO, "could not be copied."
  ENDTRY
ENDREAD
```

This example opens the local IMAGE database PRODDb and opens a remote Oracle database on the system named BLUE. A user name of ORAUSeR and a password of ORAPASS are used to logon to the remote system. The remote Oracle database on the system BLUE is opened by the Warehouse server using an Oracle user name of SCOTT and an Oracle password of TIGER. All ORD records with a STAT field equal to CLOS are read from the IMAGE database. A TRY statement starts the beginning of a TRY/RECOVER block. A COPY statement with the WAIT option is used to copy each ORD record to the remote Oracle database. WAIT causes Warehouse to wait for a response from the server for each record copied. If there is an error copying the record, the RECOVER block is entered and the ORDNO for the ORD record is printed. Without the WAIT incorrect order numbers will be printed in the RECOVER block.

#### CREATE

The CREATE statement is used to create a remote file across a network. The remote system must be running a Warehouse server program and the an AUTHFILE entry must exist on the remote system that permits remote access from your local system.

#### Syntax

```
CREATE db-tag REMOTE remote-system
      USER=user-name [PASSWORD=password]
      [EPASS1=encrypted-password]
      file-type file-name
      [file-options]
```

db-tag is the name of the database tag used to reference the database in the remainder of the script.

remote-system is the name of the remote system on which the remote file is to be created. The remote system must be running a Warehouse server and must have an AUTHFILE entry allowing a connection from the local system. The remote system may also be specified as an IP address. By default Warehouse connects to the remote system



using port 1610. To connect to a Warehouse server running on a different port the syntax: `remote-system : port` may be used, e.g.  
`EAGLE.TAURUS.COM:32400`

`user-name` is the name of the user on the remote system. When the connection is established the server logs in as `user-name` which establishes file access and file security restrictions on the remote system. The `user-name` is also used to establish the current working directory, which is the directory the file is created in by default. There must be an `AUTHFILE` entry permitting `user-name` from your system to login to the server.

NOTE: When connecting to a remote MPE/iX system, any plain text passwords must be included in the `user-name` by putting them after a slash in the standard MPE/iX job card syntax, e.g.

`USER=MGR/UPASS.ACCT/APASS`

When using encrypted passwords on MPE/iX systems, passwords are not necessary in `user-name`.

`password` is the password for `user-name` on the remote system. A password may or may not be required depending on the `AUTHFILE` entry on the server. `PASSWORD=` may be abbreviated to simple `PASS=`.

`encrypted-password` is an encrypted password for `user-name` on the remote system. An encrypted password for use in the `CREATE` statement may be generated by running Warehouse with `-c` (See **Checking Warehouse Server Connections** in [Chapter Seven](#)) or by DataBridger Studio. Password encryption is done by a proprietary algorithm based on the Data Encryption Standard (DES). A password may or may not be required depending on the `AUTHFILE` entry on the server.

`file-type` indicates the type of file to be created on the remote system. `file-type` must be one of:

ARCHIVE	Remote Warehouse archive file
CSV	Remote comma separated file
FIXED	Remote fixed record length file
TEXT	Remote text (character) file

`file-name` is the name of the file to be created on the remote system.

`file-options` are any file options that are needed. Each file type has different options that may be specified.

Once a remote file has been created, Warehouse can access it just as if the file was actually located on the local system.

#### Examples

##### Example 1

```
CREATE CF REMOTE PURPLE &  
  USER=PURPUSER PASS=PURPPASS &  
  CSV CUSTFIL &  
  MODE=READ WRITE ERASE &  
  DELIM=; QUOTE="'" FIELDNAMES
```

A comma separated values file called CUSTFIL is created on the system PURPLE. Access to PURPLE is accomplished with the user name PURPUSER and password PURPPASS.

#### DELETE

The DELETE statement deletes the current record from a remote database. The restrictions for DELETE are the same as those for the remote file or database type.

#### Examples

##### Example 1

```
OPEN DB REMOTE GREEN &  
  USER=GRNUSER PASS=GRNPASS &  
  ORACLE ORAUSER/ORAPASS  
READ M = DB.PARTS.MASTER &  
  FOR STATUS = "NOTREF"  
DELETE M  
ENDREAD
```

This example opens a remote Oracle database on the system GREEN using an Oracle user name of ORAUSER and a Oracle password of ORAPASS. The connection is established to GREEN using a user name of GRNUSER and a password of GRNPASS. The table PARTS.MASTER is read for all parts that have NOTREF in the STATUS field. All records selected from the remote table are then deleted.

**OPEN**

The OPEN statement is used to access a remote database or file across a network. The remote system must be running a Warehouse server program and the an AUTHFILE entry must exist on the remote system that permits remote access from your local system.

**Syntax**

```
OPEN db-tag REMOTE remote-system
      USER=user-name [PASSWORD=password]
      [EPASS1=encrypted-password]
      database-type database-name
      [database-options]
```

db-tag is the name of the database tag used to reference the database in the remainder of the script.

remote-system is the name of the remote system on which the remote database resides. The remote system must be running a Warehouse server and must have an AUTHFILE entry allowing a connection from the local system. The remote system may also be specified as an IP address. By default Warehouse connects to the remote system using port 1610. To connect to a Warehouse server running on a different port the syntax: remote-system : port may be used, e.g.  
EAGLE.TAURUS.COM:32400

user-name is the name of the user on the remote system. When the connection is established the server logs in as user-name which establishes file access and file security restrictions on the remote

system. The `user-name` is also used to establish the current working directory. There must be an `AUTHFILE` entry permitting `user-name` from your system to login to the server.

NOTE: When connecting to a remote MPE/iX system, any plain text passwords must be included in the `user-name` by putting them after a slash in the standard MPE/iX job card syntax, e.g.

`USER=MGR/UPASS.ACCT/APASS`

When using encrypted passwords on MPE/iX systems, passwords are not necessary in `user-name`.

`password` is the password for `user-name` on the remote system. A password may or may not be required depending on the `AUTHFILE` entry on the server. `PASSWORD=` may be abbreviated to simple `PASS=`.

`encrypted-password` is an encrypted password for `user-name` on the remote system. An encrypted password for use in the `OPEN` statement may be generated by running Warehouse with `-c` (See **Checking Warehouse Server Connections** in Chapter Seven) or by DataBridger Studio. Password encryption is done by a proprietary algorithm based on the Data Encryption Standard (DES). A password may or may not be required depending on the `AUTHFILE` entry on the server.

`database-type` indicates the type of database to be opened on the remote system. Note that the permitted types depend on the types of databases supported on the *remote* system, not on the local system. Thus a remote `IMAGE` database may be opened on *any* type of local system, provided the remote system is an MPE/iX system running a Warehouse server. `database-type` must be one of:

<code>ALLBASE</code>	Remote Allbase database environment file on an MPE/iX system
----------------------	--

ARCHIVE	Remote Warehouse archive file
CSV	Remote comma separated file
FIXED	Remote fixed record length file
IMAGE	Remote IMAGE database on an MPE/iX system
ODBC	Remote ODBC database
ORACLE	Remote Oracle database
TEXT	Remote text (character) file

`database-name` is the name of the database to be opened on the remote system. In the case of a remote Oracle database, `file-name` would really be the Oracle user name and password.

`database-options` are any database options that are needed. Each database type has different options that may be specified.

Once a remote database has been opened, Warehouse can access it just as if the database was actually located on the local system.

### Examples

#### Example 1

```
OPEN ORD REMOTE YELLOW USER=YUSR &  
ORACLE SCOTT/TIGER &  
HOME=/ORATAUR/PUB SID=A
```

Warehouse is run on a local HP-UX system and a remote Oracle database on the system YELLOW is opened. Access to YELLOW is accomplished using the user name YUSR. No password is necessary because the AUTHFILE on YELLOW indicates that no password is necessary. The Oracle user name SCOTT, password TIGER, an Oracle home of /ORATAUR/PUB, and an Oracle system ID of A is used to open the database. A Warehouse database tag of ORD is used to reference the database in the remainder of the script.

#### Example 2

```
OPEN T REMOTE RED &
```

```
USER=DOUG PASSWORD=BLVD &  
ORACLE SCOTT/TIGER &  
HOME=/b/u01/oradata/ora SID=ora  
READ X=A.EMP  
PRINT EMPNO, ENAME, JOB, MGR  
ENDREAD  
GO
```

Opens an Oracle database on the remote system RED and prints data from the EMP table.

### Example 3

```
OPEN B REMOTE BLUE.TAURUS.COM &  
USER=MGR.ACCT/APASS &  
IMAGE SALES.DATABASE PASS=; MODE=5  
READ C = B.COMPANY FOR &  
STATUS = "GC" ORDER BY COMPANY-NAME  
PRINT COMPANY-KEY, COMPANY-NAME  
ENDREAD
```

Opens a TurboIMAGE database called SALES on the remote system BLUE.TAURUS.COM and prints data from the COMPANY dataset. Note the local system can be a Unix system, Windows NT system, or any system running Warehouse.

### Example 4

```
OPEN ORD REMOTE YELLOW USER=YUSR &  
EPASS1=4768f72a55bc34a004cfe109f79e7 &  
ORACLE SCOTT &  
EPASS1=a55a204cf109f59e7f8162c5c9220 &  
HOME=/ORATAUR/PUB SID=ORAA
```

A remote Oracle database on the system YELLOW is opened. Access to YELLOW is accomplished using the user name YUSR and an encrypted password. The Oracle user name SCOTT is used with an encrypted password. An Oracle home of /ORATAUR/PUB, and an Oracle system ID of ORAA is used to open the database.

READ

The READ statement is used to read records from remote database tables and remote files. The restrictions for READ are the same as those for the remote file or database type.

## Syntax

```
READ read-tag = remote-file
    [FORMAT format-name]
    [FOR condition]
    [ORDER BY order-list]
```

`remote-file` is the identifier of the remote file from which the data is to be read. For a remote database, this is typically in the format of `db-tag.table-name`. For a remote file, this is simply `db-tag`.

When a `FORMAT` is specified, `format-name` overrides the actual definition of the table with the field names and types coming from the `format-name`.

When a `FOR` condition is specified, Warehouse selects only those records matching `condition`. If no `FOR` condition is specified, all records are selected.

When `ORDER BY` is specified, Warehouse orders (sorts) the records as specified by `order-list`.

## Examples

Example 1

```
OPEN SALES REMOTE BLACK &
    USER=HAZEL PASSWORD=BEAUTY &
    ORACLE SCOTT/TIGER
READ M = SALES.CUST.MASTER &
    FOR STATUS = "CLOS" &
    ORDER BY CUST_NAME
    READ D = SALES.CUST.ORDERS &
    FOR CUST_NO = M.CUST_NO
    PRINT CUST_NO, M.CUST_NAME, ORD_AMT
ENDREAD
ENDREAD
```

This example opens the Oracle database on the remote system `BLACK`. The user name `HAZEL` and password `BEAUTY` is used to establish the connection. The Oracle database is opened using a user name of `SCOTT` and a user password of `TIGER`. A database tag `SALES` is used to access this database environment in the remainder of the

script. The customer master table CUST.MASTER is read and records where the STATUS field has a value of CLOS are selected. All qualifying records from CUST.MASTER are ordered by the value in the CUST\_NAME field. For each CUST.MASTER record, detail records from the CUST.ORDERS table are read using the indexed field CUST\_NO. For each record from CUST.ORDERS, the fields, CUST\_NO and ORD\_AMT are printed along with CUST\_NAME from the CUST.MASTER table.

**SET**

The SET statement is used to set access options for the remote database or file.

**Syntax**

SET db-tag db-option value

db-tag specifies the database tag of the remote database or file to which the SET statement is to apply.

db-option specifies the name of the option to be changed. The db-option and value depends on the type of the remote database or file. The details for SET are contained in this chapter under each database.

**Examples****Example 1**

```
OPEN C REMOTE VIOLET USER=USR &  
      ORACLE SYSTEM/MANAGER  
SET C LONGSIZE 250000
```

This example opens a remote Oracle database on the system VIOLET using a user name of USR. Oracle is accessed using a user name SYSTEM and a user password MANAGER. The database tag C is used to access the database in the remainder of the script. The LONGSIZE is set to 250,000 allowing up to 250,000 characters to be read into or written from LONG and LONG RAW fields.

**UPDATE**

The UPDATE statement updates a field within a



record read from a remote database or file. The restrictions for `UPDATE` are the same as those for the remote file or database type.

### Examples

#### Example 1

```
OPEN C REMOTE OLIVE &  
  USER=SAM PASSWORD=HOUSTON &  
  ORACLE JONES/AAA  
CREATE A ARCHIVE ARFILE  
READ M = C.DB.MAST FOR STATUS = "CLOS"  
  COPY M TO A.DB.MAST  
  UPDATE M SET STATUS = "ARCH"  
ENDREAD
```

This example opens a remote Oracle database on the system `OLIVE` using a user name of `SAM` and a password of `HOUSTON`. Oracle is accessed with a user name `JONES` and a user password `AAA`. The database tag `C` is used to access the database in the remainder of the script. An archive file called `ARFILE` is created using the database tag `A`. The table `DB.MAST` is read and records where the `STATUS` field has a value equal to `CLOS` are selected. All qualifying records from `DB.MAST` are copied to the archive file, and the field `STATUS` is updated to the value of `ARCH`.

## REPORT

## Report file access

This section describes the considerations for each Warehouse statement that can access a report file.

Report files are special Warehouse files that are accessed by the `HEADER` and `PRINT` statements. To direct report output to other than the standard output, a report file needs to be opened with the `OPEN` statement, and the `HEADER` and `PRINT` statements need to direct output to the report file.

## CREATE

The `CREATE` statement is not supported for report files; the `OPEN` statement is used instead.

## DELETE

The `DELETE` statement is not supported for report files.

## HEADER

The `HEADER` statement is used to specify a page header for a report file.

## Syntax

```
HEADER [rpt-tag] [header-list]
```

`rpt-tag` is the file tag of the report file tag as specified in the `OPEN` statement. Note that the `rpt-tag` must be enclosed in square brackets ([ ]).

`header-list` is a list of strings or expressions to be displayed at the top of every report page.

## Examples

Example 1

```
OPEN REP REPORT RPTFIL
HEADER [REP] $TAB 60, "PAGE", $PAGENO
HEADER [REP] $CENTER, "TAURUS SOFTWARE"
```

This example opens `RPTFIL` with a tag of `REP` to be used throughout the script. The two `HEADER` statements create a two line page header. The first line contains the page number; the second line contains `TAURUS SOFTWARE` centered in the page.

OPEN	The OPEN statement is used to access a report file.
Syntax	<p>OPEN rpt-tag REPORT file-name [CCTL]</p> <p>rpt-tag is the database tag used to reference the report file in the remainder of the script.</p> <p>file-name is the file name of the report file to be opened.</p> <p>CCTL is used to indicate that Fortran style carriage control is to be used to control pagination in the report file. For MPE/iX users, a :FILE equation specifying CCTL needs to be issued for the report file to get proper pagination.</p>
Examples	<p><u>Example 1</u></p> <pre>:FILE RPTFIL;DEV=LP;CCTL OPEN REP REPORT RPTFIL CCTL HEADER [REP] \$CENTER, "TAURUS SOFTWARE"</pre> <p>This example issues an MPE/iX :FILE equation for the file RPTFIL, pointing it at the line printer and specifying that carriage control is to be used. The OPEN statement then opens RPTFIL with a database tag of REP to be used throughout the script. CCTL is specified on the OPEN statement to indicate that carriage control is to be used on the report file. A HEADER statement is directed to the output file REP.</p>
READ	The READ statement is not supported for report files.
SET	The SET statement is used to set the report page length and the report page width options.
Syntax	<p>SET rpt-tag report-option value</p> <p>rpt-tag specifies the tag of the report file to which the SET statement is to apply.</p>

`report-option` specifies the name of the option to be changed. `report-option` must be one of the following for report files:

`PAGELength` Sets the length of the report page in lines. The default page length is 55 lines. A page length of 0 causes no automatic page breaks.

`PAGEWidth` Sets the width of the report page in columns. The default page width is 80 columns. The maximum page width is 1024.

### Examples

#### Example 1

```
OPEN R REPORT MYLP
SET R PAGELength 59
SET R PAGEWidth 132
```

This example opens an report file `MYLP` with no carriage control. The database tag `R` is used to access the report file in the remainder of the script. The page length is set to 59 lines, and the page width is set to 132 columns.

### UPDATE

The `UPDATE` statement is not supported for report files.

**TEXT****Text file access**

This section describes the considerations for each Warehouse statement that accesses text files.

Text file access is specified using the type `TEXT` in the `OPEN` or `CREATE` statement and is supported on all Warehouse platforms.

Text files should be used primarily when reading or writing character data. When reading or writing binary data, a `FIXED` file should probably be used.

**COPY...TO**

When copying to a text file, the file must have been opened for write access. This is typically done with the `CREATE` statement.

**Examples****Example 1**

```
OPEN ARCH ARCHIVE ARCFIL
CREATE ORFIL TEXT ORDFILE
READ ORDS = ARCH.ORDS FOR DATE = 920222
  COPY ORDS TO ORFIL
ENDREAD
```

This example opens the archive file `ARCFIL`, and creates a new text file `ORDFILE` which is given a database tag `ORFIL`. All `ORDS` records from the archive file with a `DATE` field equal to 920222 are read and are then copied into the file `ORDFILE`.

**CREATE**

The `CREATE` statement is used to create a new text file.

**Syntax**

```
CREATE file-tag TEXT file-name
      [MODE=mode]
```

`file-tag` is the file tag used to reference the file in the remainder of the script.

`file-name` is the file name of the file to be created.

`mode` is the mode the file is to be accessed in. The mode definitions are as follows:

READ	Read access to file.
WRITE	Write access to file. ERASE or READ must be specified with WRITE. (On MPE systems, WRITE may be specified without ERASE or READ.)
APPEND	Append access to file. WRITE or ERASE not permitted with APPEND access.
BINARY	File contains BINARY data. Implementation of this parameter depends on the operating system. It is recommended whenever accessing a file with binary data.
ERASE	Erases contents of file prior to access. If ERASE is specified, file is not required to already exist. If erase is not specified, file is required to already exist.
COMMIT	(Windows only.) Contents of the file buffer are written directly to disk when <code>fflush()</code> is called.
VAR	(MPE only) Do not internally buffer records. Assume each read and write is consists of one record. This is necessary to read variable length records.
EXCLUSIVE	(MPE and Warehouse message files only) Access the file exclusively.
SHARE	(MPE only) Access the file in shared mode.
LOCK	(MPE only) Access the file with locking enabled.
TRANS	Access the file in transaction protected mode. Using TRANS

allows COMMIT and ROLLBACK by keeping all file changes in memory buffers until a commit or rollback is done. A commit writes the buffers to disk, and a rollback discards the buffers.

MSG	File is message file. On MPE systems, this is an MPE message file. On other platforms, this is a Warehouse message file.
NDR	File is an MPE message file to be read with non-destructive reads. On MPE, specifying NDR implies MSG too.
CLIB	(MPE only) Use C library instead of MPE intrinsic calls to access files.
UPDATE	(MPE only) Access the file with update access. Use of UPDATE is necessary to delete or update KSAM files.

If mode is not specified, the default mode of WRITE ERASE for write access is used.

## Examples

### Example 1

```
CREATE ORD TEXT ORDFILE
```

This example creates the text file ORDFILE for write access using the file tag ORD to access the file in the remainder of the script.

### Example 2

```
CREATE TMP TEXT TEMP &  
    MODE=READ WRITE ERASE
```

This example creates the text file TEMP for read and write access using the database tag TMP to access the file in the remainder of the script.

DELETE	The DELETE statement is not supported for text files.												
OPEN	The OPEN statement is used to access a text file.												
Syntax	<p>OPEN file-tag TEXT file-name [MODE=mode]</p> <p>file-tag is the file tag used to reference the file in the remainder of the script.</p> <p>file-name is the file name of the text file to be opened.</p> <p>mode is the mode the file is to be accessed in. The mode definitions are as follows:</p> <table><tr><td>READ</td><td>Read access to file.</td></tr><tr><td>WRITE</td><td>Write access to file. ERASE or READ must be specified with WRITE. (On MPE systems, WRITE may be specified without ERASE or READ.)</td></tr><tr><td>APPEND</td><td>Append access to file. WRITE or ERASE not permitted with APPEND access.</td></tr><tr><td>BINARY</td><td>File contains BINARY data. Implementation of this parameter depends on the operating system. It is recommended whenever accessing a file with binary data.</td></tr><tr><td>ERASE</td><td>Erases contents of file prior to access. If ERASE is specified, file is not required to already exist. If erase is not specified, file is required to already exist.</td></tr><tr><td>COMMIT</td><td>(Windows only.) Contents of the file buffer are written directly to disk when fflush() is called.</td></tr></table>	READ	Read access to file.	WRITE	Write access to file. ERASE or READ must be specified with WRITE. (On MPE systems, WRITE may be specified without ERASE or READ.)	APPEND	Append access to file. WRITE or ERASE not permitted with APPEND access.	BINARY	File contains BINARY data. Implementation of this parameter depends on the operating system. It is recommended whenever accessing a file with binary data.	ERASE	Erases contents of file prior to access. If ERASE is specified, file is not required to already exist. If erase is not specified, file is required to already exist.	COMMIT	(Windows only.) Contents of the file buffer are written directly to disk when fflush() is called.
READ	Read access to file.												
WRITE	Write access to file. ERASE or READ must be specified with WRITE. (On MPE systems, WRITE may be specified without ERASE or READ.)												
APPEND	Append access to file. WRITE or ERASE not permitted with APPEND access.												
BINARY	File contains BINARY data. Implementation of this parameter depends on the operating system. It is recommended whenever accessing a file with binary data.												
ERASE	Erases contents of file prior to access. If ERASE is specified, file is not required to already exist. If erase is not specified, file is required to already exist.												
COMMIT	(Windows only.) Contents of the file buffer are written directly to disk when fflush() is called.												



VAR	(MPE only) Do not internally buffer records. Assume each read and write consists of one record. This is necessary to read variable length records.
EXCLUSIVE	(MPE and Warehouse message files only) Access the file exclusively.
SHARE	(MPE only) Access the file in shared mode.
LOCK	(MPE only) Access the file with locking enabled.
TRANS	Access the file in transaction protected mode. Using TRANS allows COMMIT and ROLLBACK by keeping all file changes in memory buffers until a commit or rollback is done. A commit writes the buffers to disk, and a rollback discards the buffers.
MSG	File is message file. On MPE systems, this is an MPE message file. On other platforms, this is a Warehouse message file.
NDR	File is an MPE message file to be read with non-destructive reads. On MPE, specifying NDR implies MSG too.
CLIB	(MPE only) Use C library instead of MPE intrinsic calls to access files.
UPDATE	(MPE only) Access the file with update access. Use of UPDATE is necessary to delete or update KSAM files.

If mode is not specified, the default mode of READ

WRITE for read and write access is used. See the OPEN statement under FIXED files form information about file open modes.

### Examples

#### Example 1

```
OPEN ORD TEXT ORDFILE
```

This example opens the text file ORDFILE for read access using the file tag ORD to access the file in the remainder of the script.

#### Example 2

```
OPEN TMP TEXT TEMP MODE=READ WRITE
```

This example opens the text file TEMP for read and write access using the file tag TMP to access the file in the remainder of the script.

### READ

The READ statement is used to read records from text files.

### Syntax

```
READ read-tag = db-tag  
    FORMAT format-name  
    [FOR condition]  
    [ORDER BY order-list]
```

FORMAT is used to specify the record format of the file. FORMAT is required to specify the record layout of text files.

When a FOR condition is specified, Warehouse selects only those records matching condition. If no FOR condition is specified, all records in the file are selected.

When ORDER BY is specified, Warehouse orders the records as specified by order-list.

### Examples

#### Example 1

```
OPEN SALES TEXT SALESFIL MODE=READ  
OPEN SDB IMAGE SALES.DB PASS=MYPASS  
FORMAT SALESFIL_FMT
```

```
CO-NUM : X8
CO-NAME : X40
END
READ F = SALES FORMAT SALESFIL_FMT
  READ C = SDB.ORDERS &
    FOR CO-NUM = F.CO-NUM
      PRINT CUST-NO, F.CUST-NAME, ORD-AMT
    ENDREAD
  ENDREAD
```

This example opens the text file SALESFIL using the database tag SALES and the mode of `r` for read access. The IMAGE database SALES.DB is also opened using a database tag of SDB and a password of MYPASS. The record format SALESFIL\_FMT is defined with the two fields CO-NUM and CO-NAME. SALESFIL file is read using the format SALESFIL\_FMT with all records being selected. Matching orders are then read from the ORDERS dataset in SALES.DB. For each record from CUST.ORDERS, the fields CUST-NO and ORD-AMT are printed along with CO-NAME from the SALESFIL file.

SET

There are no special SET options for text files.

UPDATE

The UPDATE statement is not supported for text files.

**XML****XML file access**

This section describes the considerations for each Warehouse statement that accesses XML files.

**Syntax**

```
OPEN db-tag XML xml-file-name  
    [MODE=mode] [CASEID] [NSTRING]  
    [FLATTEN] [COLATTR]
```

`xml-file-name` is the name of the XML file to open.

`mode` is the open mode of the file. Same as `FIXED`, `TEXT` and `CSV` files.

`CASEID` Indicates that the identifiers in the XML file are case sensitive. Using `CASEID` may require you to specify table and column names in braces.

`NSTRING` Indicates that Warehouse use the `NSTRING` type columns for accessing the XML file. By default, all columns in the XML file are of `STRING` type. If any columns may contain wide (16-bit) characters, the `NSTRING` parameter should be specified.

`FLATTEN` causes Warehouse to place additional columns in XML records that can be used to link nested records. The additional column names are in the format of `column_KEYLINK` and contain a 10 digit number linking nested records. Without using `FLATTEN` records that contain nested records are unavailable in Warehouse.

`COLATTR` causes XML file attributes to be interpreted as columns. Using `COLATTR` is the only method of making attributes accessible using Warehouse.

The internal XML parser built in Warehouse can parse most any XML file, but Warehouse is only designed to handle row and column type structures. This means that only XML suitable for interpretation as rows and columns can be accessed

with Warehouse. If an XML is not suitable for Warehouse, an error is issued on the OPEN statement. To see the Warehouse interpretation of an XML file, use the SHOW statement.

XML files may also be written using the CREATE and COPY statements.

## Examples

### Example 1

```
OPEN SRC ODBC
CREATE TGT XML CUSTFIL.XML
READ C=SRC.CUSTOMERS
      COPY C TO TGT.CUSTOMERS
ENDREAD
```

### Example 2

The XML file below contains nested records.

```
<?xml version="1.0"?>
<dataroot>
  <aaa>
    <af1>1 AAA Field 1</af1>
    <af2>1 AAA Field 2</af2>
    <bbb>
      <bf1>1.1 BBB Field 1</bf1>
      <bf2>1.1 BBB Field 2</bf2>
      <ccc>
        <cf1>1.1.1 CCC Field 1</cf1>
        <cf2>1.1.1 CCC Field 2</cf2>
      </ccc>
    </bbb>
    <bbb>
      <bf1>1.2 BBB Field 1</bf1>
      <bf2>1.2 BBB Field 2</bf2>
      <ccc>
        <cf1>1.2.1 CCC Field 1</cf1>
        <cf2>1.2.1 CCC Field 2</cf2>
      </ccc>
    </bbb>
  </aaa>
</dataroot>
```

Without using FLATTEN, only the CCC data is available with Warehouse because CCC is the only record that does not contain nested records. When FLATTEN is specified, the DATAROOT, AAA, BBB,

and CCC tables all become available with layouts as follows:

XML DATAROOT layout:

DATAROOT\_KEYLINK      Parent Link to  
                                 nested records (AAA)

Table AAA layout:

DATAROOT\_KEYLINK      Child Link back  
                                 to parent (DATAROOT)

AF1

AF2

AAA\_KEYLINK              Parent Link to  
                                 nested records (BBB)

Table BBB layout:

AAA\_KEYLINK              Child Link back  
                                 to parent (BBB)

BF1

BF2

BBB\_KEYLINK              Parent Link to  
                                 nested records (CCC)

Table CCC layout:

BBB\_KEYLINK              Child Link back  
                                 to parent (BBB)

CF1

CF2

The data from this file is as follows:

Record #1: DATAROOT

DATAROOT\_KEYLINK : 0000000001

Record #2: AAA

DATAROOT\_KEYLINK : 0000000001

AF1 : 1 AAA Field 1

AF2 : 1 AAA Field 2

AAA\_KEYLINK : 0000000002

Record #3: BBB

AAA\_KEYLINK : 0000000002

BF1 : 1.1 BBB Field 1

BF2 : 1.1 BBB Field 2

BBB\_KEYLINK : 0000000003

Record #4: CCC

BBB\_KEYLINK : 0000000003

CF1 : 1.1.1 CCC Field 1

CF2 : 1.1.1 CCC Field 2

```

Record #5: BBB
    AAA_KEYLINK      : 0000000002
    BF1              : 1.2 BBB Field 1
    BF2              : 1.2 BBB Field 2
    BBB_KEYLINK      : 0000000005

Record #6: CCC
    BBB_KEYLINK      : 0000000005
    CF1              : 1.2.1 CCC Field 1
    CF2              : 1.2.1 CCC Field 2

```

### Example 3

Here are some examples with the sample XML file called WEATHER.XML

```

<?xml version="1.0"?>
<WEATHERREPORT
    xmlns="WeatherSchema.xml">
  <STATE STATENAME="California">
    <CITY CITYNAME="Los Angeles">
      <SKIES>Partly cloudy</SKIES>
      <HI>87</HI>
      <LOW>65</LOW>
    </CITY>
  </STATE>
  <STATE STATENAME="Nevada">
    <CITY CITYNAME="Las Vegas">
      <SKIES>Sunny</SKIES>
      <HI>98</HI>
      <LOW>74</LOW>
    </CITY>
  </STATE>
</WEATHERREPORT>

```

Open weather XML with default parameters and only CITY is available:

```
OPEN W XML WEATHER.XML
```

```

Tables available:
  CITY

```

```

Layout of CITY:
  SKIES
  HI
  LOW

```

Open weather XML with the COLATTR parameter and only CITY is available, but the CITYNAME attribute can be accessed:

```
OPEN W XML WEATHER.XML COLATTR
```

Tables available:  
CITY

Layout of CITY:  
SKIES  
HI  
LOW  
CITYNAME

Open weather XML with the FLATTEN parameter and WEATHERREPORT, STATE and CITY are available with KEYLINKS linking them, however STATENAME and CITYNAME are not available because they are attributes within the XML file:

```
OPEN W XML WEATHER.XML FLATTEN
```

Tables available:  
WEATHERREPORT  
STATE  
CITY

Layout of WEATHERREPORT:  
WEATHERREPORT\_KEYLINK (Parent Link)

Layout of STATE:  
WEATHERREPORT\_KEYLINK (Child Link)  
STATE\_KEYLINK (Parent Link)

Layout of CITY:  
STATE\_KEYLINK (Child Link)  
SKIES  
HI  
LOW

Open weather XML with the both the FLATTEN and COLATTR parameters. This makes WEATHERREPORT, STATE and CITY are available with KEYLINKS linking them. STATENAME and CITYNAME are also available as columns:



OPEN W XML WEATHER.XML FLATTEN COLATTR

Tables available:

WEATHERREPORT  
STATE  
CITY

Layout of WEATHERREPORT:

XMLNS  
WEATHERREPORT\_KEYLINK (Parent Link)

Layout of STATE:

WEATHERREPORT\_KEYLINK (Child Link)  
STATENAME  
STATE\_KEYLINK (Parent Link)

Layout of CITY:

STATE\_KEYLINK (Child Link)  
SKIES  
HI  
LOW  
CITYNAME



**Chapter Five**  
**Warehouse Expressions**

### Chapter Overview

This chapter describes in detail how expressions are used in Warehouse statements. Warehouse expressions can be used in conjunction with many Warehouse statements including: `IF`, `READ`, `SETVAR`, `PRINT`, `UPDATE`, and `WHILE`.

**Identifiers**

Identifiers used in Warehouse statements are used to reference many types of objects. An identifier may reference a database tag, read tag, record, array, field, or variable. Warehouse identifiers may be up to 80 characters long, consisting of the following characters:

- A . . Z Alphabetic.
- a . . z Alphabetic. Lower case is upshifted.
- 0 . . 9 Numeric digits.
- \_ Underscore
- + Plus sign
- Minus sign
- \* Asterisk
- / Slash
- ? Question mark
- # Pound sign
- % Percent sign
- & Ampersand
- @ At sign
- ' Apostrophe

Note that identifiers may contain upper and lower case characters, but lower case characters are always upshifted internally.

**Identifiers with lower case and special characters**

It is possible to create and access Warehouse identifiers that contain lower case and special characters. Variables containing lower case or special characters must be enclosed in curly braces { }. The following examples define and print three variables called "abc", "My Var", and "A~B".

```
DEFINE {abc} : INTEGER
DEFINE {My Var} : STRING
DEFINE {A~B} : STRING
PRINT {abc}, {My Var}, {A~B}
```

**Database tags**

A database tag is an identifier created with either the CREATE or OPEN statement. A file within the database may be referenced by using the db-tag and file-name separated by a dot as in: db-tag.file-name. For example:

```
OPEN DBTAG IMAGE MYDB
READ C = DBTAG.MYFILE
```

**Read tags**

A read tag identifier is created with the `READ` statement and is used both to perform operations on the file and to access fields within the file record. (It is not necessary to specify the read tag to access fields within the active `READ` statement.) The following example reads `MYFILE` from the database `MYDB` using a `READ` tag of `MYTAG`, and both `FLD1` and `FLD2` from `MYFILE` are printed.

```
READ MYTAG = MYDB.MYFILE
  * Print FLD1 without using MYTAG
PRINT FLD1
  * Print FLD2 using MYTAG
PRINT MYTAG.FLD2
DELETE MYTAG
ENDREAD
```

**Record identifiers**

Records identifiers are created with either the `DEFINE` statement or the `READ` statement. Fields within the record are accessed using the record name and field name separated by a dot as in: `record-name.field-name`. Example:

```
DEFINE CUST : RECORD
  CUST-NO      : X8
  NAME         : X40
  ADDRESS      : RECORD
    ADDR1      : X40
    ADDR2      : X40
    CITY       : X12
    ST         : X2
    ZIP        : X10
  END
END

SETVAR CUST.CUST-NO      = "00000000"
SETVAR CUST.NAME        = " "
SETVAR CUST.ADDRESS.ADDR1 = " "
SETVAR CUST.ADDRESS.ADDR2 = " "
...
```

**Array identifiers**

Arrays are created with the `DEFINE` statement and they may also be contained within records read from a file. Array elements are indexed using the array name with the array index enclosed in square

brackets as in: array-name[array-index].

Example:

```
DEFINE IX : 12
DEFINE TOTALS : 1212
SETVAR IX = 1
WHILE IX <= 12
    TOTALS[IX] = 0
    SETVAR IX = IX + 1
ENDWHILE
```

**Constants**

Warehouse constants are used within expressions. There are two types of constants, string and numeric. In general, a string constant is enclosed in quotes and a numeric constant is just the number. There are also system constants that are special variables that begin with a dollar sign (\$).

**Numeric Constants**

The format of numeric constants is:

`[+|-] digits [.digits]`

`digits` represents an unsigned number consisting of any number of digits from 0 through 9. Numeric constants may have an optional sign, and a decimal point followed by decimal digits.

Numeric constants are unsigned unless preceded by a plus or minus sign. The distinction between signed and unsigned numbers is usually not important; however, when reading a file by key, the sign of the key may be significant.

Examples of valid numeric constants

12345	unsigned 12,345
-54321	negative 54,321
+2.38	positive 2.38
.0004	0.0004

Examples of invalid numeric constants

12,345	contains comma
7FFF	contains alpha characters
1.23E+6	contains alpha characters
664-	has trailing sign

**String Constants**

String constants are specified by enclosing a string of characters in either single (') or double (") quotes. Quotations may be continued by putting more than one quotation successively with spaces between. The backslash (\) can be used to place certain special characters in a string constant. The following backslash combinations represent a single character within a string:



\ "	Double quote
\ '	Single quote
\\	Backslash
\a	Bell
\b	Backspace
\n	New line
\r	Carriage return
\t	Tab

Examples:

String	Result
'Taurus'	Taurus
'With "quotes"'	With "quotes"
'I shouldn\t'	I shouldn't
"\Help\" "	"Help"
"one\two"	one\two
"\aSTOP!\a"	<bell>STOP!<bell>
"one" "two"	onetwo

### System Constants

Warehouse provides a number of system constants. The constants are listed below:

Name	Type	Description
\$ERR	Record	Error information
\$FALSE	Boolean	False
\$HOUR	Numeric	Time in HHMMSS format
\$MYPID	Integer	Warehouse operating system process id (PID).
\$NOW	DateTime	Current date / time
\$NOW0	DateTime	Current date / time
\$NULL	Null	Null value
\$RECNUM	Other	IMAGE database column name
\$TODAY	Numeric	Date in YYYYMMDD
\$TRUE	Boolean	True
\$UNKNOWN	Null	Unknown value

\$MYPID is an integer constant that contains the Warehouse operating system process id (PID). This can be useful for certain situations, such as writing log files. Example:

```
PRINT "My pid=", $MYPID
```

`$NOW` is a `DATETIME` system variable that returns the current date and time and is manipulated by Warehouse as a *date*. `$TODAY` and `$HOUR` are numeric variables that contain the date and time and are manipulated by Warehouse as *numbers*. `$NOW` is recalculated every time it is accessed by the script.

`$NOW0` is the same as `$NOW`, except that `$NOW0` is calculated only once by Warehouse and remains the same throughout the entire script run. For example, `$NOW0` can be used to timestamp records all records with the same timestamp when doing a load.

Note: The time zone may need to be set correctly for `$HOUR`, `$NOW`, `$NOW0` and `$TODAY` to return the correct current values. On MPE/iX and Unix operating systems, this is done by setting the `TZ` operating system environment variable. For example, on MPE/iX the following command is issued outside of Warehouse to set the time zone to Pacific time:

```
SETVAR TZ "PST8PDT"
```

`$RECNUM` is a virtual column name. For more information see [Chapter 4, IMAGE File Types](#), `READ` and `SET` commands.

Examples	Expression	Result
	<code>(1 = 2) = \$FALSE</code>	True
	<code>\$TODAY - 19000000</code>	980908
	<code>STR(STRING(\$TODAY), 3, 6)</code>	"980908"
	<code>\$TODAY</code>	19980908
	<code>\$HOUR</code>	141245
	<code>\$ERR.WHERRNO</code>	8001

### **\$ERR System Variable**

When performing error recovery, it is often necessary to know what type of error occurred, the error number, the error message, or possibly other

error information. Warehouse uses a system variable called `$ERR` to contain this information. `$ERR` is a record variable with the following fields:

<code>DBERRNO</code>	: <code>INTEGER</code>
<code>DBERRMSG</code>	: <code>STRING</code>
<code>ERRTYPE</code>	: <code>STRING</code>
<code>ESCMSG</code>	: <code>STRING</code>
<code>WHERRMSG</code>	: <code>STRING</code>
<code>WHERRNO</code>	: <code>INTEGER</code>
<code>LINENO</code>	: <code>INTEGER</code>

`DBERRNO` contains the error number of the most recent database or file operation. When `DBERRNO` is set, `WHERRNO` and `WHERRMSG` are also set.

`DBERRMSG` contains the error message text of the most recent database or file error. This is associated with `DBERRNO`. A zero value indicates no database error has occurred.

`ERRTYPE` is a string that describes the type of error most recently encountered by Warehouse.

`ERRTYPE` is one of:

<code>ALLBASE</code>	Most recent error is an Allbase database error file.
<code>ARCHIVE</code>	Most recent error is an archive file I/O error.
<code>CSV</code>	Most recent error is a CSV file I/O error.
<code>FIXED</code>	Most recent error is a fixed file I/O error.
<code>IMAGE</code>	Most recent error is a TurboIMAGE database error.
<code>ORACLE</code>	Most recent error is an Oracle database error.
<code>TEXT</code>	Most recent error is a text file I/O error.
<code>WHERR</code>	Most recent error is a Warehouse error.
<code>WHWARN</code>	Most recent error is a Warehouse warning.

`ESCMSG` contains the string set by the most recent

ESCAPE statement.

WHERRMSG contains the error message text of the most recent Warehouse error. This is associated with WHERRNO.

WHERRNO contains the error number of the most recent Warehouse error. A zero value indicates that no error has occurred.

LINENO is used to display the script line number on which the most recent error has occurred.

To give the user more control handling errors, the \$ERR error record has both read and write access.

### Examples

```

OPEN SALES IMAGE SALES.DB PASS=READER MODE=5
OPEN ORSAL REMOTE SYS04 USER=user PASS=myspass &
  ORACLE SCOTT/TIGER &
  HOME=/oradata/ora/bin SID=orcl

DEFINE STATUS : ORACLE CHAR(1) VALUE "Y"

TRY
  READ M = SALES.CUSTMAST
  COPY M TO ORSAL.CUSTOMERS
  TRY
    READ D = SALES.CUSTHIST &
      FOR CUSTNO = M.CUSTNO
    COPY D TO ORSAL.CHISTORY
  ENDREAD
  RECOVER
    IF $ERR.ERRTYPE = "ORACLE" AND &
      $ERR.DBERRNO = 1
      * Back out incomplete history
    READ OD = ORSAL.CHISTORY FOR &
      CUSTNO = M.CUSTNO
    DELETE OD
  ENDREAD
  ELSE
    ESCAPE "*** Unrecognized error"
  ENDIF
  ENDTRY
  ENDREAD
  RECOVER
    SETVAR STATUS = "N"
  ENDTRY

  IF STATUS = "Y"
    PRINT "All customers successfully copied"
  ELSE
    PRINT "***** Error copying customers"
  ENDIF

```

In this script, customer records are copied from a local TurboIMAGE database on an HP3000 to a remote Oracle database. The TRY statement begins a TRY block that encloses the outer READ statement. After the CUSTMAST record is copied, another TRY block is entered which reads customer history records and copies them to the Oracle database. If there is an error while copying the history records, a RECOVER block is entered that checks to make certain the error is an Oracle error, and that the Oracle error number is 1. If the error is Oracle error 1, all history records that were copied are backed out. If the error wasn't an Oracle error, or wasn't error 1, an ESCAPE is issued to the outer RECOVER.

**Operators**

The two types of operators supported are unary and binary. Unary operators only operate on a single operand to the right of the operator. Binary operators operate on the two operands on either side of the operator.

**Unary Operators**

Unary operators supported are: unary positive (+) and unary negative (-) and NOT for Boolean values.

+	Unary positive
-	Unary negative
BNOT	Bitwise NOT
NNOT	Logical NOT on a boolean expression
NOT	Logical NOT

Unary positive may only be used on numeric data types. For data types that do not support unsigned data, unary positive returns the value of the operand. For data types that support unsigned data, unary positive returns the value of the operand if it is signed. If the operand is unsigned, the positive value of the operand is returned.

Unary negative may only be used on numeric data types. For negative values, the positive value of the operand is returned. For positive and unsigned values, the negative value of the operand is returned.

The BNOT operator does a bitwise not operation on an integer value. BNOT operates on integers of arbitrary length. Negative numbers represent an arbitrary number of 1 bits and non-negative numbers represent an arbitrary number of 0 bits.

The NNOT operator is used to negate a logical or Boolean value. NNOT returns true for a null operand.

The NOT operator is used to negate a logical or Boolean value.

**Unary Operator Examples**

<u>Expression</u>	<u>Result</u>
-------------------	---------------

NOT (1 > 2)	\$TRUE
-(2 - 5)	3
BNOT 2	-3
BNOT -3	2

## NNOT Example

Given these definitions:

```

DEFINE A : ODBC CHAR(10) ALLOW NULLS
VALUE "A"
DEFINE B : ODBC CHAR(10) ALLOW NULLS
VALUE "B"
DEFINE N : ODBC CHAR(10) ALLOW NULLS
VALUE $NULL

```

The results are:

Expression	Result
NNOT (A < B)	False
NNOT (A < N)	True
NOT (A < B)	False
NOT (A < N)	Unknown

## Binary Operators

Binary operators require two operands. The following operators are supported:

.	Record field selector
[ ]	Array index
*	Multiplication
/	Division
+	Addition
-	Subtraction
	String concatenation
=	Equal
<>	Not equal
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal
==	Equal to with null equal
AND	Logical AND
BAND	Bitwise AND
BOR	Bitwise OR
BSL	Bit shift left
BSR	Bit shift right
BXOR	Bit exclusive OR
DIVF	Floating point divide

DIVI	Integer divide
OR	Logical OR
MOD	Modulus (Remainder)
PIC	Print Picture

The `.` operator is used to select a field within a record.

The `[ ]` operator is used to index into an array using the syntax:

```
array-name [ index-value ]
```

The bit operators `BSL`, `BSR`, `BAND`, `BOR` and `BXOR` operate on integers of arbitrary length and return an integer of arbitrary length. Negative numbers represent an arbitrary number of 1 bits and non-negative numbers represent an arbitrary number of 0 bits.

The arithmetic operators (`+`, `-`, `/`, `*`, `DIVF`, `DIVI`, `MOD`) only operate on numeric data types and result in numeric results.

The `/` operator performs division on two operands and does either a floating point division or an integer division depending on the types of the operands. It is often difficult to determine ahead of time which type of division will be used. Therefore the use of the `/` is discouraged and either the `DIVF` operator (floating point divide) or the `DIVI` operator (integer divide) should be used instead of `/`.

The `||` operator concatenates two strings. The `+` operator may also be used to concatenate strings, but its use is discouraged because of potential confusion with `+` to do addition.

The comparison operators (`=`, `<>`, `<`, `>`, `<=`, `>=`, `==`) operate on numeric and string data types and return a Boolean result.

When comparing fields that may be null, care must be taken to choose the correct comparison operator.



The equal to operator (=) returns *unknown* when one or both of the operands are null. This in accordance with SQL standards and maintains compatibility between Warehouse and SQL databases. Two equal signs (==) can be used to compare two operands and return *true* when both operands are null. The == operator returns false when one operand is null and the other is not, otherwise the == operator returns the same result as the = operator.

The logical operators AND and OR operate on Boolean data.

The PIC operator is used to format numeric or date data. See the PRINT statement in ***Warehouse Statements*** for more information on PIC.

#### Array comparison

Comparison for equality or inequality (=, <>) of arrays is permitted. To compare two arrays, the array bounds must be identical and the array elements must be in the same type family.

When comparing two arrays that with elements allowing nulls, if an element is null in both the arrays, the two elements are considered *equal* to each other. This is contrary to the result when comparing the two null elements directly in which case the result is considered unknown.

#### Record comparison

Comparison for equality or inequality (=, <>) of records is permitted. To compare two records, the records must be identically structured. To compare records that are of similar, but not identical structure the CONVERT or MAGICON function may be used to convert one of the records to the same type as the other. When comparing records, two fields that are null are considered equal.

When comparing two records that contain fields allowing nulls, if a field is null in both the records, the two fields are considered *equal* to each other. This is contrary to the result when comparing the two null fields directly in which case the result is

considered unknown.

#### Order of Precedence

Expressions are evaluated from left to right in order of operator precedence. Parentheses may be used to override the default order of precedence. Operators at the same level of precedence are evaluated left to right. The order of precedence is as follows:

##### Highest

1 .	Record field
[ ]	Array index
2 . BSL	Bit shift left
BSR	Bit shift right
3 . +	Unary positive
-	Unary negative
BNOT	Bitwise not
NOT	Logical not
4 . *	Multiplication
/	Division
BAND	Bitwise AND
DIVF	Floating point division
DIVI	Integer division
MOD	Modulus (remainder)
5 . +	Addition
-	Subtraction
	String concatenation
BOR	Bitwise OR
BXOR	Bitwise exclusive OR
6 . <	Less than
<=	Less than or equal to
=	Equal to
>	Greater than
>=	Greater than or equal to
<>	Not equal to
==	Equal to with null equal

7. AND Logical AND

8. OR Logical OR

9. PIC Print picture

### Character Set Conversion

#### Lowest

When operations with strings of differing character sets are performed, an automatic character set conversion is done using the CMAP function

The operations that can generate an automatic CMAP are:

Comparison operators: <, <=, =, >=, >, <>, ==

String assignment: SETVAR statement, UPDATE statement

String concatenation: ||

Strings may or may not have a character set. When a string operation is performed, no character set conversion is done if either string has no character set or if the strings have the same character set. Conversion is only done when both strings have a character set and the two character sets differ.

Automatic character set conversion may be overridden using the CONVERT or FIELD functions using a target type with no character set or a different character set.

For example, the following two code snippets have the result in the same value in TGTNAM:

#### Snippet 1:

```
DEFINE SRCNAM : IMAGE X8
DEFINE TGTNAM : ODBC CHAR(8)
SETVAR TGTNAM = CMAP(SRCNAM, "HP-
ROMAN8", "ISO8859-1")
```

#### Snippet 2:

```
DEFINE SRCNAM : IMAGE X8 CHARSET
```

```
"HP-ROMAN8"
DEFINE TGTNAM : ODBC CHAR(8)
CHARSET "ISO8859-1"
SETVAR TGTNAM = SRCNAM // Auto
CMAP done here
```

## Examples

Expression	Result
5 * 2 + 1	11
1 + 5 * 2	11
(1 + 5) * 2	12
13 MOD 5	3
2 + 13 MOD 5	5
"AB"    " "    "CDE"	"AB CDE"
"PN"    (74 PIC "9(6)")	"PN000074"
MYARRAY[1 + 2 * 3]	MYARRAY[7]
14 DIVF 5	2.8
14 DIVI 5	2
BNOT 2 BSL 1	-5
(BNOT 2) BSL 1	-6

## Implicit Type Conversion

When evaluating an expression of mixed data types, Warehouse often needs to convert intermediate results to a common data type. For example if an ORACLE NUMBER item is added to a SQL DECIMAL item, each is first converted to a Warehouse NUMERIC, and then the addition is done.

When evaluating an expression Warehouse requires each operand to belong to a compatible type family and Warehouse implicitly converts each operand to the appropriate intermediate data type. An error is issued if there is an attempt to use an operand from an incompatible type family. For example, it is an error to attempt to add string data with a numeric data even if the string contains valid numeric characters. To do an explicit type conversion, the SETVAR statement may be used to store an intermediate result, or the CONVERT function may be used.

Care must be exercised when using fixed length string types such as CHAR, IMAGE X, ORACLE CHAR, and ODBC CHAR. By default Warehouse

strips trailing spaces from a fixed length character data type when converting to a variable length character type such as `STRING`, `ORACLE VARCHAR2`, and `ODBC VARCHAR`. This conversion often happens implicit and sometimes does not have the desired result. The `PAD` function may be used in an expression to prevent trailing spaces from being stripped automatically, or `AUTOPAD` can be set to prevent trailing spaces from being stripped in the entire script.

**Date Operators**

Date operators are used to perform operations on date data. In general: two dates/times may be subtracted to produce an interval, a date/time and an interval may added to produce another date/time, dates/times may be compared with each other.

**Date Addition**

An INTERVAL may be added to a DATE or DATETIME resulting in a DATETIME. An INTERVAL may also be added to a TIME or another INTERVAL resulting in an INTERVAL. The following table shows the result of date addition:

DATETIME	+	DATETIME	=	Error
DATETIME	+	DATE	=	Error
DATETIME	+	TIME	=	DATETIME
DATETIME	+	INTERVAL	=	DATETIME
DATE	+	DATETIME	=	Error
DATE	+	DATE	=	Error
DATE	+	TIME	=	DATETIME
DATE	+	INTERVAL	=	DATETIME
TIME	+	DATETIME	=	DATETIME
TIME	+	DATE	=	DATETIME
TIME	+	TIME	=	INTERVAL
TIME	+	INTERVAL	=	INTERVAL
INTERVAL	+	DATETIME	=	DATETIME
INTERVAL	+	DATE	=	DATETIME
INTERVAL	+	TIME	=	INTERVAL
INTERVAL	+	INTERVAL	=	INTERVAL

**Date Subtraction**

A DATE or DATETIME may be subtracted from another DATE or DATETIME resulting in an INTERVAL. Two INTERVALS or two TIMES may also be subtracted resulting in an INTERVAL. The following table shows the result of date subtraction:

DATETIME	-	DATETIME	=	INTERVAL
DATETIME	-	DATE	=	INTERVAL
DATETIME	-	TIME	=	DATETIME
DATETIME	-	INTERVAL	=	DATETIME
DATE	-	DATETIME	=	INTERVAL
DATE	-	DATE	=	INTERVAL
DATE	-	TIME	=	DATETIME
DATE	-	INTERVAL	=	DATETIME
TIME	-	DATETIME	=	Error
TIME	-	DATE	=	Error

TIME	-	TIME	=	INTERVAL
TIME	-	INTERVAL	=	INTERVAL
INTERVAL	-	DATETIME	=	Error
INTERVAL	-	DATE	=	Error
INTERVAL	-	TIME	=	INTERVAL
INTERVAL	-	INTERVAL	=	INTERVAL

### Date Comparison

Date comparison is done using the operators: <, <=, =, >=, >, and <>. A date comparison is legal between DATES, and DATETIMES, or between TIMES and INTERVALS, but not between DATES/DATETIMES and TIMES/INTERVALS. The following table shows the legal date compares based on the date subtype:

Date1	Date2	Is Legal?
DATETIME	DATETIME	Yes
DATETIME	DATE	Yes
DATETIME	TIME	No
DATETIME	INTERVAL	No
DATE	DATETIME	Yes
DATE	DATE	Yes
DATE	TIME	No
DATE	INTERVAL	No
TIME	DATETIME	No
TIME	DATE	No
TIME	TIME	Yes
TIME	INTERVAL	Yes
INTERVAL	DATETIME	No
INTERVAL	DATE	No
INTERVAL	TIME	Yes
INTERVAL	INTERVAL	Yes

**Null Operations**

Null operations are operations where one of the operands contains a null value. In general when an operation or function is performed on a null value, the result is null.

**Operations on nulls**

Whenever one of the unary operators NOT, -, or + is used on a null value the result is null. Whenever one of the binary operators \*, /, MOD, -, +, or || is used on a null value the result is null.

Unless otherwise indicated, Warehouse functions also return null when a null value is used as an operand.

**Null Comparisons**

Whenever one of the comparison operators <, <=, =, >=, >, or <> is used on a null, the result is considered *unknown*. An exception to this is when two fields containing null are compared within a record or array comparison, they are considered equal. (The Warehouse constant \$UNKNOWN can be used to indicate an unknown value.)

The == operator can be used to return *true* when both operands are null. The == operator returns *false* when one operand is null and the other is not. The == operator returns the same as = when neither value is null

**Examples**

Assuming the following Warehouse statements have been executed to define and assign values to 4 variables:

```
DEFINE ISNULL1      : ORACLE NUMBER ALLOW NULLS
DEFINE ISNULL2      : ORACLE NUMBER ALLOW NULLS
DEFINE ISNTNULL1    : ORACLE NUMBER ALLOW NULLS
DEFINE ISNTNULL2    : ORACLE NUMBER ALLOW NULLS

SETVAR ISNULL1 = $NULL
SETVAR ISNULL2 = $NULL
SETVAR ISNTNULL1 = 8
SETVAR ISNTNULL2 = 3
```

Then the following expressions will result in the value indicated:



Expression	Result
-ISNULL1	\$NULL
-ISNTNULL1	-8
ISNTNULL1 + ISNULL2	\$NULL
ISNTNULL1 + ISNTNULL2 + ISNULL2	\$NULL
ABS(ISNULL1)	\$NULL
ISNULL1 = ISNULL2	\$UNKNOWN
IF(ISNULL1 = ISNULL2, "YES", "NO")	"NO"
ISNULL1 <> ISNULL2	\$UNKNOWN
IF(ISNULL1 <> ISNULL2, "YES", "NO")	"NO"
ISNULL1 = ISNTNULL2	\$UNKNOWN
ISNULL1 <> ISNTNULL2	\$UNKNOWN
IF(ISNULL1 <> ISNTNULL2, "YES", "NO")	"NO"
ISNULL1 < ISNTNULL2	\$UNKNOWN
ISNULL1 == ISNTNULL2	\$FALSE
ISNULL1 == ISNULL2	\$TRUE
NOT (ISNULL1 == ISNULL2)	\$FALSE

**Built-In Functions**

Warehouse supports built-in functions to do many different operations, such as date and string manipulations.

**ABS**

Returns the absolute value of a number. If the data type of the parameter supports unsigned values, ABS returns an unsigned value, otherwise a positive value is returned.

**Usage**

`ABS(number) -> number`

**Examples**

<u>Expression</u>	<u>Result</u>
<code>ABS(-11)</code>	11
<code>ABS(12)</code>	12

**ACCEPT**

Reads a string from the user's standard input (stdin) after displaying a prompt string.

Warning: ACCEPT is operating system and environment dependent.

**Usage**

`ACCEPT(prompt-string) -> string`

**Examples**

<u>Expression</u>	<u>Result</u>
<code>ACCEPT("Order number:")</code>	user input

**BOOLEAN**

Converts a string to a Boolean value. A valid Boolean string must be an integer value with zero being false and any other value being true, or the string must be a partial match of one of the following words: YES, NO, TRUE, FALSE, \$TRUE, \$FALSE.

**Usage**

`BOOLEAN(string) -> boolean`

string is the string to be converted to a Boolean value.

**Examples**

<u>Expression</u>	<u>Result</u>
-------------------	---------------

BOOLEAN( "NO" )	False
BOOLEAN( "T" )	True
BOOLEAN( "TR" )	True
BOOLEAN( "    Yes    " )	True
BOOLEAN( "    0001" )	True
BOOLEAN( "\$TRUE" )	True
BOOLEAN( "    -687" )	True
BOOLEAN( "+000" )	False
BOOLEAN( " N" )	False
BOOLEAN( "AB" )	Error
BOOLEAN( "    " )	Error
BOOLEAN( "-1.2" )	Error
BOOLEAN( "1e3" )	Error
BOOLEAN( "\$ TRUE" )	Error
BOOLEAN( "\$YES" )	Error
BOOLEAN( "NOPE" )	Error

**CHR** Returns the ASCII character given a numeric value from 0 to 255. To convert a character into a numeric value, use the ORD function.

**Usage** CHR(number) -> character

Examples	Expression	Result
	CHR( 65 )	"A"
	CHR( 97 )	"a"
	CHR( 400 )	Error

**CMAP** Translate characters from one character set to another using a charmap file. The charmap files used for the translation must be listed in the CHARMAPS file. See Appendix C for more information on setting up charmap files.

**Usage** CMAP(source-str, source-charset, target-charset) -> string

source-str is the string to be translated into the target character set.

source-charset is the name of the source character set as it is defined in the charmap file. This must be a constant string enclosed in quotation marks.

`target-charset` is the name of the target character set as it is defined in the charmap file. This must be a constant string enclosed in quotation marks.

`string` is the resulting translated string of characters in the `target-charset`.

#### Example

#### Expression

```
CMAP(MYSTR, "ASCII", "EBCDIC")
```

Translates `MYSTR` from ASCII to EBCDIC using charmap files.

#### CONVERT

Converts an expression to a specific data type. The `CONVERT` function requires two parameters. The first parameter is an expression, and the second parameter is a string constant that describes the data type of the expression after conversion.

Data type conversion converts the expression to the specified data type. An error can occur if the expression cannot be converted. For example, the string `12A4` cannot be converted to a number because it contains non-numeric characters.

When converting string types to numeric types, the requirements for the string are the same as when using the `NUMERIC` function: Leading and trailing spaces are permitted, a leading sign is permitted and a decimal point is permitted.

The `CONVERT` function may be used to convert a record from one layout to another. Conversion is done using the same rules as `COPY` and `SETVAR`. Fields with matching names in the record are converted and fields not in the destination record are initialized. If `MSGs` are on, then information about field conversions are displayed.

When converting numbers to strings, the results are the same as when using the `STRING` function.

Usage `CONVERT(expr, "data-type") -> result`

Examples	Expression	Result
	<code>CONVERT("1234", "IMAGE I2")</code>	1234
	<code>CONVERT("1234", "ORACLE CHAR(4)")</code>	"1234"
	<code>CONVERT("1234", "ORACLE CHAR(2)")</code>	"12"
	<code>CONVERT("12", "ORACLE CHAR(4)")</code>	"12 "
	<code>CONVERT(1432, "ALLBASE FLOAT")</code>	1432.0
	<code>CONVERT("7F", "INTEGER")</code>	Error
	<code>CONVERT(REC1, "FORMAT REC2FMT")</code>	Record
	<code>CONVERT(CUST, "USING DB.CUST")</code>	Record

DATE2STR Converts a date, time, datetime, or interval to a string using a date format.

DATE2STR is the inverse function of STR2DATE (which converts a string to a date).

Usage `DATE2STR(date, fmt-string) -> string`

`date` is the date, time, datetime, or interval type item to be converted to a string.

`fmt-string` is a string containing tokens that describe the format of `date-string`. The tokens are the same as those used in the print PIC of a date item and the DATE2STR function. The allowable tokens in `fmt-string` are:

A.D.	BC/AD indicator with periods
AD	BC/AD indicator
A.M.	AM/PM indicator with periods
AM	AM/PM indicator
AY	Two character year and century where 00-99 is century 19, and A0-J9 is century 20. e.g. A5 represents 2005
B.C.	BC/AD indicator with periods
BC	BC/AD indicator
CC	2 digit century
D	The day of week (1-7, Sun=1,Sat=7)
DAY	The 9 character name of day of week

	(SUNDAY-SATURDAY)
DD	The 2 digit day number within month (1-31)
DDD	The 3 digit day number within year (1-366)
D*	The 1 or 2 digit day number within month (1-31)
DY	The 3 character name of day of week (SUN-SAT)
HH	The 2 digit hour in 12 hour time (01-12)
HH12	The 2 digit hour in 12 hour time (01-12)
HH24	The 2 digit hour in 24 hour time (00-23)
H*	The 1 or 2 digit hour in 12 hour time (1-12)
H*12	The 1 or 2 digit hour in 12 hour time (1-12)
H*24	The 1 or 2 digit hour in 24 hour time (0-23)
J	Julian day number since January 1, 4713 BC.
MI	The 2 digit minute within the hour (00-59)
MM	The 2 digit month number within year (01-12)
M*	The 1 or 2 digit month number within year (1-12)
MON	The 3 character name of month (JAN-DEC)
MONTH	The 9 character name of month (JANUARY-DECEMBER)
NNN...	Number of days (up to 9 Ns)
P.M.	AM/PM indicator with periods
PM	AM/PM indicator
Q	Quarter within year (1-4)
RM	Roman numeral month number within year (I-XII) (4 characters)
RR	The last 2 digits of the year. (Same as YY.)
SCC	2 digit century with leading - for BC dates

SS	The 2 digit second within the minute (00-59)
SSSSS	The 5 digit second within the day (0-86399)
SYYYY	4 digit year with leading sign: + for AD dates,- for BC dates
TTT...	Fractions of seconds (up to 9 Ts)
W	The week within the month (1-5)
WW	The 2 digit week within the year (01-53)
Y	The last digit of the year
YY	The last 2 digits of the year.
YYY	The last 3 digits of the year
YYYY	The 4 digit year
Y,YYY	Year with comma
space	Space
:	Colon
/	Virgule
-	Hyphen
.	Period
,	Comma
;	Semicolon
"str"	Quoted string

Date format items are case insensitive, except that the case determines the appearance of alphabetic date items.

### Examples

The following examples assume the following statements:

```
DEFINE DTM : DATETIME
SETVAR DTM = &
DATE2STR("960401 1504","YYMMDD HH24MI")
```

Expression	Result
DATE2STR(DTM,"YYMMDD")	960401
DATE2STR(DTM,"DD-MON-YY")	1-APR-96
DATE2STR(DTM,"DD-Mon-YY")	1-Apr-96
DATE2STR(DTM,"HH24:MI:SS")	15:04:00
DATE2STR(DTM,"Month DD,YY")	April 1,96
DATE2STR(DTM,"Day, Mon DD")	Mon, Apr 1
DATE2STR(DTM,"M*/D*/YY")	4/1/96

**DAYNUM** Calculates the day number since Monday, December 31, 1900 given a numeric date in YYYYMMDD format. DAYNUM and YYYYMMDD are used to do date calculations. The DAYNUM function returns negative values for dates prior to December 31, 1900.

DAYNUM is the inverse function of YYYYMMDD. DAYNUM can be used to calculate the day of the week by taking DAYNUM MOD 7 where Monday is 0, Tuesday is 1, ..., Sunday is 6.

**Usage** DAYNUM(yyyymmdd-number) -> day-number

Examples	<u>Expression</u>	<u>Result</u>
	DAYNUM(19890704)	32327
	DAYNUM(19890704) MOD 7	1
	DAYNUM(17760704)	-45469
	DAYNUM(19010101)	1
	DAYNUM(19001230)	-1
	DAYNUM(19940931)	Error

**DIRECT** Directly executes a database statement. The DIRECT function is database dependent and is used to execute SQL statements when accessing an SQL database. The DIRECT function may only be called using the CALL statement.

The DIRECT *function* is used to execute a database statement after the GO. The DIRECT *statement* is used to execute a database statement before the go.

To trap errors returned by the DIRECT function, use the TRY statement.

**Usage** CALL DIRECT(db-tag, string)

Examples	<u>Statement</u>
	CALL DIRECT(CUSTDB, "DROP TABLE TMP")

**DWNS** Downshifts a string. DWNS converts all uppercase



characters in a given string to lowercase characters.

To upshift a string use the `UPS` function.

Usage `DWNS(string) -> string`

Examples	<u>Expression</u>	<u>Result</u>
	<code>DWNS("Taurus")</code>	<code>"taurus"</code>
	<code>DWNS("SOFTWARE")</code>	<code>"software"</code>

**ESCAPE**

Generates an error condition. The error may be caught by an active `TRY` function or `TRY` statement. If there is no active `TRY`, then an error message is displayed and script processing halts. The `ESCAPE` function operates just like the escape statement except that it may be used in an expression. The `ESCAPE` function does not return a value.

Usage `ESCAPE(error-msg)`

Examples	<u>Expression</u>
	<code>IF(ISNUMERIC(ORDNUM), NUMERIC(ORDNUM), ESCAPE("ORDNUM not numeric"))</code>

Checks if `ORDNUM` can be converted to a number. If it can then the converted number is returned, otherwise an error condition is generated.

**FIELD**

Used to extract data from a record or variable and coerce it to a specific data type. The `FIELD` function requires three parameters. The first parameter is a variable name, the second parameter is a byte index into the first parameter, and the third parameter is a string constant that describes the data type of the expression after extraction.

Usage `FIELD(var, index, "data-type") -> result`

`var` can be a defined variable, a record name, a field from a record, or the result of a `CONVERT` function. Note: You must be familiar with the data type of `var` to get the desired results.

`index` is the byte index of the first byte of data to be extracted. The first byte is byte 1.

`data-type` is a string constant that describes the field after extraction. `FIELD` simply extracts the bytes, then interprets them as the specified type. No conversion is done.

### Examples

The following examples assume the script:

```
DEFINE R : RECORD
    F    : ORACLE CHAR(4)
    G    : ORACLE CHAR(6)
END
SETVAR R.F = "ABCD"
SETVAR R.G = "efghij"
```

Expression	Result
<code>FIELD(R, 1, "ORACLE CHAR(3)")</code>	"ABC"
<code>FIELD(R, 3, "ALLBASE CHAR(3)")</code>	"CDe"
<code>FIELD(R.F, 1, "IMAGE X3")</code>	"ABC"
<code>FIELD(R.F, 3, "ORACLE CHAR(3)")</code>	Error <sup>(1)</sup>
<code>FIELD(R.G, 3, "ORACLE CHAR(3)")</code>	"ghi"
<code>FIELD(R, 3, "ALLBASE SMALLINT")</code>	17220 <sup>(2)</sup>
<code>FIELD(R, 1, "IMAGE Z1")</code>	+1 <sup>(3)</sup>

(1) Field overflow. `R.F` is only 4 characters long, so taking 4 characters starting at character 3 overflows the field.

(2) The characters CD interpreted as a 2-byte signed integer. The same as:

```
ORD("C") * 256 + ORD("D")
```

(3) A is a positive 1 in a zoned field.

### FILL

Fills string on the left. `FILL` repeatedly inserts `fill-string` to the left of `source-string` until the string is of length `final-length`. If `final-`

length is less than the length of source-string, characters are stripped from the RIGHT of source-string to make it length final-length.

Usage `FILLL(source-string,  
          final-length,  
          fill-string) -> string`

Examples	Expression	Result
	<code>FILLL("May", 6, "*")</code>	<code>"***May"</code>
	<code>FILLL("May", 8, "*-")</code>	<code>"*-*-May"</code>
	<code>FILLL("12.3", 8, "0")</code>	<code>"000012.3"</code>
	<code>FILLL("Taurus", 4, "*")</code>	<code>"urus"</code>

**FILLR** Fills string on the right. FILLR repeatedly inserts fill-string to the right of source-string until the string is of length final-length. If final-length is less than the length of source-string, characters are stripped from the LEFT of source-string to make it length final-length.

Usage `FILLR(source-string,  
          final-length,  
          fill-string) -> string`

Examples	Expression	Result
	<code>FILLR("May", 6, "*")</code>	<code>"May***"</code>
	<code>FILLR("May", 8, "*-")</code>	<code>"May*-*-"</code>
	<code>FILLR("12.3", 8, "0")</code>	<code>"12.30000"</code>
	<code>FILLR("Taurus", 4, "*")</code>	<code>"Taur"</code>

**GETENV** Returns the value of an environment variable when passed the variable name. If there is no variable of the name given, a null string is returned.

Warning: GETENV is operating system dependent.

Usage `GETENV(string) -> string`

Examples	Expression	Result
	<code>GETENV("USER")</code>	<code>"jones"</code>

**HASH** Returns a "random" 32-bit integer from the parameters. HASH returns a value created from the parameters by using a 32-bit cyclic redundancy check (CRC) algorithm. The purpose of HASH is to create a key value that can quickly be accessed using an indexed table. This can create dramatic performance improvements versus searching for a number of values individually. HASH returns a value from -2147483648 to 2147483647.

**Usage** `number = HASH(parm1 [, parm2 [, parm3 ...] ] )`

Examples	Expression	Result
	HASH(0)	-148897096
	HASH(0, 0)	-538414411
	HASH(0, 0, 0)	1309568012
	HASH(0, 1)	-618041598
	HASH(1, 0)	59107330
	HASH("One")	-982575239
	HASH("One ")	-1632807640
	HASH("One", 1)	-1743613981
	HASH(\$null)	79764919

**IF** Evaluates a Boolean expression and returns either of two expressions depending on the result of the Boolean expression.

**Usage** `IF(cond,true-value,false-value) -> value`

`cond` is any Boolean expression. The result of `cond` determines whether the IF function returns `true-value` or `false-value`.

`true-value` is an expression that is returned if `cond` evaluates to TRUE.

`false-value` is an expression that is returned if `cond` evaluates to FALSE. `false-value` must have the same data type family as `true-value`.

Examples	Expression	Result
----------	------------	--------

```

IF(5 < 9, "Yes", "No")      "Yes"
IF(3 < 7 and 9 < 5, 1, 0)    0
IF(ISNUMERIC("ab"), 10, 0)  0

```

**ISBOOLEAN**

Tests if a string can be converted to a Boolean value. If the string can be successfully converted to a Boolean value TRUE is returned, otherwise FALSE is returned. If TRUE is returned, then the BOOLEAN function can convert the string to a Boolean value without error. A valid Boolean string must be an integer value, or a partial match of one of the following words: YES, NO, TRUE, FALSE, \$TRUE, \$FALSE.

**Usage**

ISBOOLEAN(string) -> boolean

string is the string to be tested for a Boolean value.

**Examples**

<u>Expression</u>	<u>Result</u>
ISBOOLEAN("NO")	True
ISBOOLEAN("T")	True
ISBOOLEAN("TR")	True
ISBOOLEAN(" Yes ")	True
ISBOOLEAN(" 0001")	True
ISBOOLEAN("\$TRUE")	True
ISBOOLEAN(" -687")	True
ISBOOLEAN("+000")	True
ISBOOLEAN(" N")	True
ISBOOLEAN("AB")	False
ISBOOLEAN(" ")	False
ISBOOLEAN("-1.2")	False
ISBOOLEAN("1e3")	False
ISBOOLEAN("\$ TRUE")	False
ISBOOLEAN("\$YES")	False
ISBOOLEAN("NOPE")	False

**ISDATE**

Tests if a string can be converted to a date value. If the string can be successfully converted to a date value TRUE is returned, otherwise FALSE is returned. If TRUE is returned, then the STR2DATE function can convert the string to a date value without error.

## Usage

ISDATE(date-str) -> boolean

or

ISDATE(date-str, format-str) -> boolean

string is the string to be tested for a date value.

format-str is optional and if present is a string containing tokens that describe the format of date-str. See the STR2DATE function for information on allowable format strings.

If format-str is absent, ISDATE examines the string and returns TRUE if the string is in a recognizable format that can be converted to a date using the STR2DATE function. ISDATE can automatically recognize dates, times, datetimes, and intervals. See the STR2DATE function for information on how dates are interpreted.

## Examples

Expression	Result
ISDATE("960401", "YYMMDD")	True
ISDATE("12/12/24", "YY/MM/DD")	True
ISDATE("121224", "RRMMDD")	True
ISDATE("121224", "HHMISS")	True
ISDATE("Sep 8,92", "MON DD,YY")	True
ISDATE(" 960401", "YYMMDD")	False
ISDATE("12-12-24", "YY/MM/DD")	False
ISDATE("16:12:24", "HH24:MI:SS")	True
ISDATE("16:12:24", "HH:MI:SS")	False
ISDATE("960401")	True
ISDATE("12/12/24")	True
ISDATE("12/12/1924")	True
ISDATE("121224")	True
ISDATE("12-Dec-24")	True
ISDATE("Sep 8,92")	True
ISDATE(" 960401")	True
ISDATE("12-12-24")	True
ISDATE("16:12:24")	True
ISDATE("16")	True
ISDATE("May")	False

## ISDIGITS

Scans a string and returns true if all characters in the string are "0"-"9". If the string contains any characters other than "0"-"9" (including spaces) false is returned. A minimum string length may be specified. When a minimum length is specified,

false is returned if the length of the string is less than minimum. A zero length string returns false, unless a minimum length of zero was specified.

## Usage

```
result = ISDIGITS(source-string [,
                    minimum-length] )
```

source-string is the string to be checked for only digits.

minimum-length is an optional parameter that indicates the minimum length of the string. This parameter is often necessary when checking fixed length strings because spaces are truncated from a fixed length string prior to the ISDIGITS check.

## Examples

Example	Result
ISDIGITS("3141")	TRUE
ISDIGITS(" 3141")	FALSE
ISDIGITS("-3141")	FALSE
ISDIGITS("31.41")	FALSE
ISDIGITS("3141", 6)	FALSE
ISDIGITS("7F")	FALSE
ISDIGITS("")	FALSE
ISDIGITS("", 0)	TRUE

## ISNUMERIC

Tests if a string can be converted to a numeric value. If the string can be successfully converted to a numeric value TRUE is returned, otherwise FALSE is returned. If TRUE is returned, then the NUMERIC function can convert the string to a numeric value without error.

A numeric string may contain a decimal point, an E followed by an exponent, a single leading or trailing sign, a sign of CR or DB, and a comma as a 1000s separator. Leading and trailing spaces, asterisks (\*), and dollar signs (\$) are ignored. A numeric string may not contain more than one sign, more than one decimal point, or a misplaced comma separator. A null string or a string containing all spaces is interpreted as zero.

## Usage

```
ISNUMERIC(string) -> boolean
```

string is the string to be tested for a numeric

value.

Examples	Expression	Result
	ISNUMERIC("25")	True
	ISNUMERIC(" -16 ")	True
	ISNUMERIC("1.64")	True
	ISNUMERIC(" ")	True
	ISNUMERIC("1.64E+04")	True
	ISNUMERIC("-16.4E-4")	True
	ISNUMERIC("- 21")	True
	ISNUMERIC("-000021")	True
	ISNUMERIC("12,345,678")	True
	ISNUMERIC("12,345.678")	True
	ISNUMERIC("\$1.64")	True
	ISNUMERIC("-\$1.64")	True
	ISNUMERIC("44 CR")	True
	ISNUMERIC("55 DB")	True
	ISNUMERIC("1,2345.67")	False
	ISNUMERIC("1.64E+04.2")	False
	ISNUMERIC("44 AB")	False
	ISNUMERIC("-44 CR")	False
	ISNUMERIC("-44-")	False

## ISNUMP

Determines if a string is a valid packed decimal string that can be converted using the NUMP function. To be a valid packed decimal string, all nibbles (4-bits) except the last nibble must contain a binary value from 0 through 9.

The last nibble must contain one either 12, 13, or 15 as follows:

Decimal	Hexadecimal	Description
12	C	Number is positive
13	D	Number is negative
15	F	Number is unsigned (treated as positive)

## Usage

`result = ISNUMP(source-string)`

`source-string` is the string to be checked be a valid packed decimal string.

Examples	Expression	Result
	ISNUMP(CHR(\$00)    CHR(\$00))	FALSE
	CHR(\$00))	



```

ISNUMP(CHR($00) || CHR($00)
      || CHR($0F))      TRUE
ISNUMP(CHR($12) || CHR($34)
      || CHR($56))      FALSE
ISNUMP(CHR($12) || CHR($34)
      || CHR($5D))      TRUE
ISNUMP(CHR($12) || CHR($CC)
      || CHR($5C))      FALSE
ISNUMP(CHR($01) || CHR($20)
      || CHR($0C))      TRUE

```

**ISNUMZ**

Determines if a string is a valid zoned decimal string that can be converted using the NUMZ function.

To be a valid zoned decimal string, all positions except the last must contain "0"-"9". The last position of the string must contain one of:

```

"0" - "9" : indicating an unsigned number
"{"       : positive, with last digit of 0.
"A" - "I" : positive, with last digit 1-9 (A=1, I=9)
"}"       : negative, with last digit of 0.
"J" - "R" : negative, with last digit 1-9 (J=1,
          R=9)

```

**Usage**

```
result = ISNUMZ(source-string)
```

`source-string` is the string to be checked be a valid zoned decimal string.

**Examples**

Expression	Result
ISNUMZ("3141")	TRUE
ISNUMZ(" 3141")	FALSE
ISNUMZ("-3141")	FALSE
ISNUMZ("31.41")	FALSE
ISNUMZ("7F")	TRUE
ISNUMZ("7Z")	FALSE
ISNUMZ("}")	TRUE
ISNUMZ(" ")	FALSE

**LEN**

Returns the length of a string. Leading and trailing blanks are counted as part of the string length. Leading blanks may be stripped with the TRIML function, and trailing blanks may be stripped with the TRIMR function. LEN may be used on native (double byte) character strings to

return the number of characters in the string.

Usage `LEN(string) -> number`

Examples	<u>Expression</u>	<u>Result</u>
	<code>LEN(" ABCD ")</code>	9
	<code>LEN(TRIMR(" ABCD " ))</code>	6

## MAGICON

Converts a record to another record of similar structure for the purpose of record assignment or comparison.

MAGICON converts the source record to the destination layout by matching up field names in three passes. Matching fields are converted to the data type of the destination field. The first pass matches field names exactly. (e.g. In the first pass CUST-NAME matches CUST-NAME, but not CUST\_NAME) The second pass matches alphanumeric characters in the field name exactly, but matches special characters with any other special character. (e.g. In the second pass CUST-NAME matches CUST\_NAME and CUST\$NAME, but not CUSTNAME) The third pass matches alphanumeric characters in the field name exactly and special characters are ignored. (e.g. In the third pass CUST-NAME matches CUSTNAME and CUST\_N-A-M-E, but not CUSTOMER-NAME) Any fields that are unmatched in the three passes are initialized.

If MSGS are on, MAGICON displays information about each field being converted.

Usage `MAGICON(rec, "FORMAT format-name")`

or

`MAGICON(rec, "USING db-tag.table-name")`

Examples

```

OPEN IMDB IMAGE CUSTDB
OPEN ORDB ORACLE SCOTT/TIGER
READ ICUST = IM.CUSTOMER
  READ OCUST = ORDB.CUSTOMER FOR &
    CUST_NO = ICUST.CUSTNO
  IF OCUST <> &
    MAGICON(ICUST, "USING ORDB.CUSTOMER")

```

```

UPDATE OCUST SET &
  CUST_NAME = ICUST.CUST_NAME, &
  CUST_ADDR = ICUST.CUST-ADDR, &
  CITY = ICUST.CITY, &
  STATE = ICUST.STATE, &
  ZIP = ICUST.ZIP
ENDIF
ENDREAD
ENDREAD

```

This example reads customer records from an IMAGE database CUSTDB, then for each customer record read, it reads the corresponding customer record from an Oracle database. The record from the Oracle database is compared with the record from the IMAGE database to see if they are different. The `MAGICON` function is used to convert the IMAGE customer record to be the same format as the Oracle customer record. The `MAGICON` function matches the fields as follows:

<u>IMAGE</u>		<u>Oracle</u>
CUSTNO	-->	CUST_NO
CUST_NAME	-->	CUST-ADDR
CITY	-->	CITY
STATE	-->	STATE
ZIP	-->	ZIP

If the records are different, then fields in the Oracle database are updated the values from the IMAGE database.

## MATCH

Matches string with a pattern. If the string matches the pattern TRUE is returned, otherwise FALSE is returned.

## Usage

`MATCH(source, pattern) -> boolean`

`source` is the string to be matched for the pattern.

`pattern` is the pattern. Special pattern characters are listed below, all other characters must match exactly. Special pattern characters:

\* Match zero or more characters.

?	Match a single character.
@	Match alphabetic character: A-Z, a-z.
#	Match single numeric digit: 0-9
[ s ]	Range or list of characters, e.g. For hexadecimal digit use [0-9a-fA-F]
!	Escape character.

Examples	Expression	Result
	MATCH( "Taurus", "T*" )	True
	MATCH( "Taurus", "*s" )	True
	MATCH( "Taurus", "*a*r*s" )	True
	MATCH( "Taurus", "#####" )	True
	MATCH( "Taurus", "??????" )	True
	MATCH( "Taurus", "*u*" )	True
	MATCH( "Taurus", "[A-Z]*" )	True
	MATCH( "Taurus", "*#" )	False
	MATCH( "Taurus", "#####" )	False
	MATCH( "12.3", "##.#" )	True
	MATCH( "12.3", "##*" )	True
	MATCH( "@", "@" )	False
	MATCH( "@", "!@" )	True
	MATCH( "Taurus", "[a-z]*" )	False

## NUMERIC

Converts a string to a number. The string to be converted may contain a decimal point, an E followed by an exponent, a single leading or trailing sign, a sign of CR or DB, and a comma as a 1000s separator. Leading and trailing spaces, asterisks (\*), and dollar signs (\$) are ignored. A numeric string may not contain more than one sign, more than one decimal point, or a misplaced comma separator. A null string or a string containing all spaces is interpreted as zero.

NUMERIC is the inverse function of STRING (which converts a number to a string).

Usage                      NUMERIC(string) -> number

Examples	Expression	Result
	NUMERIC( "25" )	25
	NUMERIC( "        -16        " )	-16
	NUMERIC( "1.64" )	1.64

NUMERIC("1.64E+04")	16400
NUMERIC("-16.4E-4")	-0.00164
NUMERIC("-21")	-21
NUMERIC("-000021")	-21
NUMERIC("12,345,678")	12345678
NUMERIC("12,345.678")	12345.678
NUMERIC("\$1.64")	1.64
NUMERIC("-\$1.64")	-1.64
NUMERIC("44 CR")	44
NUMERIC("55 DB")	-55
NUMERIC(" ")	0
NUMERIC("1,2345.67")	Error
NUMERIC("1.64E+04.2")	Error
NUMERIC("44 AB")	Error
NUMERIC("-44 CR")	Error
NUMERIC("-44-")	Error

**NUMP** Converts a packed decimal string to a number. See **ISNUMP** above for description of a packed decimal number.

If the string passed to **NUMP** is not a valid packed decimal string, a warning is issued and the result is undefined. (The **TRY** function or statement can be used to catch the warning.)

**Usage** `result = NUMP(source-string)`

`source-string` is the string to be checked be a valid packed decimal string.

Expression	Result
NUMP(CHR(\$00)    CHR(\$00)    CHR(\$00))	Error
NUMP(CHR(\$00)    CHR(\$00)    CHR(\$0F))	0
NUMP(CHR(\$12)    CHR(\$34)    CHR(\$56))	Error
NUMP(CHR(\$12)    CHR(\$34)    CHR(\$5C))	12345
NUMP(CHR(\$12)    CHR(\$34)    CHR(\$5D))	-12345
NUMP(CHR(\$01)    CHR(\$20)    CHR(\$0C))	1200

**NUMZ** Converts a zoned decimal string to a number. See

ISNUMZ above for description of a zoned decimal number. If the string passed to NUMZ is not a valid zoned decimal string, a warning is issued and the result is undefined. (The TRY function or statement can used to catch the warning.)

## Usage

```
result = ISNUMZ(source-string)
```

source-string is the string checked to be a valid zoned decimal string.

## Examples

Expression	Result
ISNUMZ ( "3141" )	3141
ISNUMZ ( "     3141" )	Error
ISNUMZ ( "7F" )	76
ISNUMZ ( "K" )	-2

## ORD

Returns the ASCII value of a character. ORD is the inverse function of CHR. When ORD is passed a string of characters, ORD operates on only the first character in the string. For single byte character strings, ORD returns a value from -128 to 127. For native (double byte) character strings, ORD returns a value from 0 to 65535.

## Usage

```
ORD(string) -> number
```

## Examples

Expression	Result
ORD ( "0" )	48
ORD ( "z" )	122
ORD ( "Taurus" )	84
ORD ( CHR ( 200 ) )	-56
ORD ( " " )	Error

## PAD

Returns a fixed length character string with trailing blanks. Normally trailing blanks are stripped from fixed length character strings before operations are performed on them. The PAD function is used to prevent trailing blanks from being stripped.

The PAD function may only be used on the following fixed length string data types:

ALLBASE CHAR  
 CHAR  
 IMAGE X  
 ODBC CHAR  
 ORACLE CHAR  
 SQL CHAR

When AUTOPAD is turned on with the SET statement, trailing blanks are always retained when performing operations on fixed length character strings and use of the PAD function is unnecessary.

Usage PAD(string) -> string

Examples The following examples assume a variable called C6 has been defined as a CHAR(6) data type and the value has been set to "ABCD".

<u>Expression</u>	<u>Result</u>
PAD(C6)	"ABCD "
LEN(C6)	4
LEN(PAD(C6))	6
C6 + C6	"ABCDABCD"
PAD(C6) + C6	"ABCD ABCD"

POS Returns the position of a search string within a source string. If the search string is null, POS returns 1. If the search string is not found, POS returns 0.

Usage POS(search-str, source-str) -> number

<u>Expression</u>	<u>Result</u>
POS(" ", "Taurus Software")	7
POS("war", "Taurus Software")	12
POS("sof", "Taurus Software")	0
POS("", "Taurus Software")	1

REPLACE Replaces all the occurrences of a search string in a source string with replacement string.

The source-string can be of any character string or wide character string type. If source-string is null, null is returned. If search-string is null, the source-string is returned. If replacement-string is null, all occurrences of search-string are stripped from source-string.

Since REPLACE is also a standard SQL function, if REPLACE is used in the FOR condition of a READ statement on an SQL database (Oracle, SQL Server, DB2), the REPLACE is passed on to the database engine with the database engine doing the evaluation of REPLACE. This may cause unexpected results since the databases may handle the corner conditions differently.

Usage `string = REPLACE(source-string, search-string, replacement-string)`

Exress	Result
REPLACE('BANANA', 'NA', 'S')	'BASS'
REPLACE('BANANA', 'A', 'N')	'BNNNNN'
REPLACE('****', '*', '***')	'*****'
REPLACE('I was here', ' ', '')	'Iwashere'
REPLACE('None', 'X', 'Y')	'None'
REPLACE('AbcAbc', 'Abcd', 'XYZ')	'AbcAbc'

### Example 7

When using REPLACE in a FOR condition so that it's passed to the database for processing has the following behavior when NULL is used for the replacement string:

On Oracle:  
`REPLACE('Taurus', 'us', null)`  
 results in 'Taur'

On SQL Server:  
`REPLACE('Taurus', 'us', null)`  
 results in null

ROUND

Rounds a number to a specified number of digits.



Usage `ROUND(number, num-digits) -> number`

`number` is the number to be rounded.

`num-digits` indicates the number of decimal digits to the right of the decimal to which `number` is to be rounded. If `num-digits` is equal to zero, `number` is rounded to the nearest integer. If `num-digits` is negative, `number` is rounded to the power of 10 indicated by negative `num-digits`.

Examples	Expression	Result
	<code>ROUND(3.14159, 4)</code>	3.1416
	<code>ROUND(3.14159, 2)</code>	3.14
	<code>ROUND(3.14159, 6)</code>	3.141590
	<code>ROUND(-3.14159, 4)</code>	-3.1416
	<code>ROUND(7419.917, 0)</code>	7420
	<code>ROUND(7419.917, -1)</code>	7420
	<code>ROUND(7419.917, -2)</code>	7400
	<code>ROUND(7419.917, -4)</code>	10000

SCRUB Searches source-string for characters less than a space (hex 20) or greater than a tilde (hex 7E) and replaces each character found with the replacement-string. To delete each binary character, use "" for the replacement-string.

Usage `SCRUB(source-str, repl-str) -> string`

Examples	Expression	Result
	<code>SCRUB("Taurus\r\n", "**")</code>	"Taurus**"
	<code>SCRUB("Taurus\r\n", "")</code>	"Taurus"

SIZEOF Returns the number of bytes required by a variable or record.

Usage `SIZEOF(expression) -> number`

Examples Assume the follow definitions:

```
DEFINE IMAGE11VAR : IMAGE I1
DEFINE ODBCINTVAR: ODBC INTEGER
DEFINE ORACHAR20VAR: ORACLE CHAR(20)
```

```

DEFINE ORAREC : RECORD
  NUM : ORACLE CHAR(10)
  NAM : ORACLE CHAR(20)
  ADR : ORACLE CHAR(40)
END

```

<u>Expression</u>	<u>Result</u>
SIZEOF( IMAGEI1VAR )	2
SIZEOF( ODBCINTVAR )	4
SIZEOF( ORACHAR20VAR )	20
SIZEOF( ORAREC )	70

**SLEEP**

Causes the script to pause for a specified number of seconds. SLEEP must be accessed with the CALL statement.

**Usage**

CALL SLEEP(seconds)

**Examples**

Statement

CALL SLEEP(600)

Causes the Warehouse script to sleep for 10 minutes.

**STR**

Extracts a substring from a source string, given a starting position and a length. The position of the first character in a string is 1.

**Usage**

STR(string, start, length) -> string

**Examples**

<u>Expression</u>	<u>Result</u>
STR("ABCD", 2, 3)	"BCD"
STR("ABCD", LEN("ABCD")-1, 2)	"CD"
STR("ABCD", 1, 0)	" "

**Example 4**

```

DEFINE FC : ODBC CHAR(12) ALLOW &
  NULLS VALUE "ABCD"
PRINT STR(FC, 3, 4)

```

Displays the following:

	"CD "																								
STR2DATE	<p>Converts a string to a date, time, datetime, or interval using a specified date format, or automatically recognizes a date if no format is specified.</p> <p>STR2DATE is the inverse function of DATE2STR (which converts a date to a string).</p>																								
Usage	<p>STR2DATE(date-str) -&gt; date</p> <p>or</p> <p>STR2DATE(date-str, format-str) -&gt; date</p> <p>date-str is a string containing the date to be converted to a date type item.</p> <p>format-str is optional and if present contains a string of tokens that describe the format of date-str. The tokens are the same as those used in the print PIC of a date item and the DATE2STR function. The allowable tokens in format-str are:</p> <table> <tr> <td>A.D.</td><td>BC/AD indicator with periods</td></tr> <tr> <td>AD</td><td>BC/AD indicator</td></tr> <tr> <td>A.M.</td><td>AM/PM indicator with periods</td></tr> <tr> <td>AM</td><td>AM/PM indicator</td></tr> <tr> <td>AY</td><td>2 Character year and century where 00-99 is century 19, and A0-J9 is century 20. e.g. A5 represents 2005</td></tr> <tr> <td>B.C.</td><td>BC/AD indicator with periods</td></tr> <tr> <td>BC</td><td>BC/AD indicator</td></tr> <tr> <td>CC</td><td>2 digit century</td></tr> <tr> <td>D</td><td>The day of week (1-7, Sun=1,Sat=7)</td></tr> <tr> <td>DAY</td><td>The 9 character name of day of week (SUNDAY-SATURDAY)</td></tr> <tr> <td>DD</td><td>The 2 digit day number within month (01-31)</td></tr> <tr> <td>DDD</td><td>The 3 digit day number within year (01-366)</td></tr> </table>	A.D.	BC/AD indicator with periods	AD	BC/AD indicator	A.M.	AM/PM indicator with periods	AM	AM/PM indicator	AY	2 Character year and century where 00-99 is century 19, and A0-J9 is century 20. e.g. A5 represents 2005	B.C.	BC/AD indicator with periods	BC	BC/AD indicator	CC	2 digit century	D	The day of week (1-7, Sun=1,Sat=7)	DAY	The 9 character name of day of week (SUNDAY-SATURDAY)	DD	The 2 digit day number within month (01-31)	DDD	The 3 digit day number within year (01-366)
A.D.	BC/AD indicator with periods																								
AD	BC/AD indicator																								
A.M.	AM/PM indicator with periods																								
AM	AM/PM indicator																								
AY	2 Character year and century where 00-99 is century 19, and A0-J9 is century 20. e.g. A5 represents 2005																								
B.C.	BC/AD indicator with periods																								
BC	BC/AD indicator																								
CC	2 digit century																								
D	The day of week (1-7, Sun=1,Sat=7)																								
DAY	The 9 character name of day of week (SUNDAY-SATURDAY)																								
DD	The 2 digit day number within month (01-31)																								
DDD	The 3 digit day number within year (01-366)																								

D*	The 1 or 2 digit day number within month (1-31)
DY	The 3 character name of day of week (SUN-SAT)
HH	The 2 digit hour in 12 hour time (01-12)
HH12	The 2 digit hour in 12 hour time (01-12)
HH24	The 2 digit hour in 24 hour time (00-23)
H*	The 1 or 2 digit hour in 12 hour time (1-12)
H*12	The 1 or 2 digit hour in 12 hour time (1-12)
H*24	The 1 or 2 digit hour in 24 hour time (0-23)
J	Julian day number since January 1, 4713 BC.
MI	The 2 digit minute within the hour (00-59)
MM	The 2 digit month number within year (01-12)
M*	The 1 or 2 digit month number within year (1-12)
MON	The 3 character name of month (JAN-DEC)
MONTH	The 9 character name of month (JANUARY-DECEMBER)
NNN. . .	Number of days (up to 9 Ns)
P.M.	AM/PM indicator with periods
PM	AM/PM indicator
Q	Quarter within year (1-4)
RM	Roman numeral month number within year (I-XII) (4 characters)
RR	The last 2 digits of the year. (The century is assumed to be 20 for years 00-49 and assumed to be 19 for years 50-99.)
SCC	2 digit century with leading - for BC dates
SS	The 2 digit second within the minute (00-59)
SSSSS	The 5 digit second within the day (0-

	86399)
SYYY	4 digit year with leading sign: + for AD dates,- for BC dates
TTT...	Fractions of seconds (up to 9 Ts)
W	The week within the month (1-5)
WW	The 2 digit week within the year (01-53)
Y	The last digit of the year
YY	The last 2 digits of the year. If no century is specified (cc), the current century is assumed.
YYY	The last 3 digits of the year
YYYY	The 4 digit year
Y,YYY	Year with comma
space	Space
:	Colon
/	Virgule
-	Hyphen
.	Period
,	Comma
;	Semicolon
"str"	Quoted string

Date format items are case insensitive in the STR2DATE function.

Items are required to be the specified length. For example the MONTH token requires a 9 character month name, so APRIL must have 4 trailing spaces.

If items are missing, default values are used as follows:

Default century:	0 if BC is specified, otherwise the current century.
Default year:	0 if century or BC is specified, otherwise the current year.
Default month:	1 (January)
Default day:	1
Default hour:	0
Default minute:	0

Default second: 0

If `format-str` is absent, `STR2DATE` examines `date-str` and converts it to a date if it is in a recognizable format. It is an error if the `date-str` is not in a recognizable format. The `ISDATE` function can be used to determine if the string is in a recognizable format. `STR2DATE` can automatically recognize dates, times, datetimes, and intervals. If a string can be either a date or a time, then a date is recognized. e.g. "121314" is interpreted as 13-Dec-2014, not 12:13:14 PM. A two digit year from 00 to 49 is interpreted to be from 2000 to 2049, and a two digit year from 50 to 99 is interpreted to be from 1950 to 1999.

Recognizable formats are:

Date formats:

MM/DD/RR  
 MM/DD/YYYY  
 MM/DD  
 MM-DD-RR  
 MM-DD-YYYY  
 MM-DD  
 DD-MON-RR  
 DD-MON-YYYY  
 DD-MON  
 RRMDD  
 YYYYMMDD  
 Month DD, RR  
 Month DD, YYYY  
 Month DD

Time formats:

HH24:MI  
 HH24:MI:SS AM, PM, A.M. or P.M. may follow  
 HH24:MI:SS.T\*  
 HH24MI

Datetime formats:

`datetime` Date in one of the above formats followed by a time in one of the above formats.

Interval formats:

N\* HH24:MI  
 N\* HH24:MI:SS  
 N\* HH24:MI:SS.T\*  
 N\* Simple number is  
 interpreted as interval

Examples	Expression	Result
	STR2DATE("960401","YYMMDD")	01-APR-1996
	STR2DATE("121224","YYMMDD")	24-DEC-1912
	STR2DATE("121224","RRMMDD")	24-DEC-2012
	STR2DATE("121224","HHMISS")	12:12:24 PM
	STR2DATE("Sep 8,92","MON DD,YY")	08-SEP-1992
	STR2DATE("960401")	01-APR-1996
	STR2DATE("12/12/24")	12-DEC-2024
	STR2DATE("12/12/1924")	12-DEC-1924
	STR2DATE("121224")	12-DEC-2024
	STR2DATE("121264")	12-DEC-1964
	STR2DATE("12-Dec-24")	12-DEC-2024
	STR2DATE("Sep 8,92")	08-SEP-1992
	STR2DATE("960401")	01-APR-1996
	STR2DATE("12-12-24")	12-DEC-2024
	STR2DATE("12:12:24")	12:12:24 PM
	STR2DATE("12")	12 Days
	STR2DATE("12 4:14")	12 Days 4 Hours 14 Minutes
	STR2DATE("9/8/92 17:21")	08-SEP-1992 5:21 PM
	STR2DATE("May")	* Error

**STRING**

Converts a number to a string.

STRING is the inverse function of NUMERIC (which converts a string to a number).

Usage                      STRING(number) -> string

Examples	Expression	Result
	STRING(25)	"25"

**SYSTEM**

Executes an operating system command. The result is operating system dependent with a result of 0 indicating successful command execution.

The SYSTEM function is different from the ! statement in that the SYSTEM function executes as part of the script, whereas the ! statement executes immediately.

Warning: SYSTEM is operating system dependent.

Usage                      SYSTEM(string) -> number

Examples	<u>Expression</u>	<u>Result</u>
	SYSTEM("SETVAR TERM,HP")	0

TOKEN                      Parses a string and returns the Nth token within the string.

Usage                      TOKEN(source-string, token-number, "delimiters-flags")

source-string is the string to be parsed.

token-number is the token number to be returned with 1 being the first token in the string. If token number is less than 1 or greater than the number of tokens in source-string a string of zero length is returned.

delimiters-flags is a string enclosed in quotation marks that indicate the parsing delimiters and flags. The delimiters may be any special characters such as comma, colon, semicolon and space. In addition to special characters the s and q flags are available.

s flag - Indicates that leading trailing spaces are stripped from the token.

q flag - Indicates that tokens may be enclosed in quotation marks.

Examples                      A = 'one;two,"three,four";five, six , seven,'  
B = ' alpha beta|gamma delta'

<u>Expression</u>	<u>Result</u>
TOKEN(A, 1, ",")	one;two
TOKEN(A, 1, ";")	one
TOKEN(A, 2, ";")	two,"three,four"
TOKEN(A, 2, ",q")	three,four
TOKEN(A, 2, ",;")	two
TOKEN(A, 4, ",;")	four"
TOKEN(A, 4, ",;q")	five



```
TOKEN(A, 5, ",;q")    six
TOKEN(A, 5, ",;qs")   six
TOKEN(B, 1, " ")
TOKEN(B, 1, " s")      alpha
TOKEN(B, 3, " s")      delta
TOKEN(B, 3, " |s")     gamma
```

**TOKENCOUNT**

Parses a string and returns the number of tokens in the string. TOKENCOUNT parses exactly like the TOKEN function, but returns the number of tokens rather than a specific token.

**Usage**

```
n = TOKENCOUNT(source-string,
                  "delimiters-flags")
```

source-string is the string to be parsed.

delimiters-flags is a constant string enclosed in quotation marks that indicate the parsing delimiters and flags. The delimiters may be any special characters such as comma, colon, semicolon and space. In addition to special characters, s and Q flags are available.

s flag - Indicates that leading and trailing spaces are stripped from the token.

Q flag - Indicates that tokens may be enclosed in quotation marks.

**Examples**

```
A = 'one;two,"three,four";five,  six  ,
    seven,'
B = '  alpha  beta|gamma  delta'
```

Expression	Result
TOKENCOUNT(A, " , ")	5
TOKENCOUNT(A, " ; ")	3
TOKENCOUNT(A, " ; ")	3
TOKENCOUNT(A, " ,q ")	5
TOKENCOUNT(A, " , ; ")	7
TOKENCOUNT(A, " , ; ")	7
TOKENCOUNT(A, " , ;q ")	6
TOKENCOUNT(A, " , ;q ")	6
TOKENCOUNT(A, " , ;qs ")	6
TOKENCOUNT(B, " ")	7
TOKENCOUNT(B, " s ")	3
TOKENCOUNT(B, " s ")	3

TOKENCOUNT(B, " |s") 4

**TRIML**

Strips leading (left) spaces. TRIML returns the string parameter with all leading blanks stripped from the front of the string. Blanks occurring after the first non-blank character are not stripped.

To strip both leading and trailing blanks, use the TRIML function combined with the TRIMR function.

**Usage**

TRIML(string) -> string

**Examples**

<u>Expression</u>	<u>Result</u>
TRIML(" To be ")	"To be "
TRIML("Or not")	"Or not"
TRIML(TRIMR(" to be "))	"to be"

**TRIMR**

Strips trailing (right) spaces. TRIMR returns the string parameter with all trailing blanks stripped from the end of the string. Blanks occurring before the last non-blank character are not stripped.

To strip both leading and trailing blanks, use the TRIML function combined with the TRIMR function.

**Usage**

TRIMR(string) -> string

**Examples**

<u>Expression</u>	<u>Result</u>
TRIMR(" To be ")	" To be"
TRIMR("Or not")	"Or not"
TRIML(TRIMR(" to be "))	"to be"

**TRUNC**

Truncates a number to a specified number of digits.

**Usage**

TRUNC(number, num-digits) -> number

number is the number to be truncated.

num-digits indicates the number of decimal digits to the right of the decimal to which number is to be truncated. If num-digits is equal to zero,

`number` is truncated to the nearest integer. If `num-digits` is negative, `number` is truncated to the power of 10 indicated by negative `num-digits`.

Examples	Expression	Result
	<code>TRUNC(3.14159, 4)</code>	3.1415
	<code>TRUNC(3.14159, 2)</code>	3.14
	<code>TRUNC(3.14159, 6)</code>	3.141590
	<code>TRUNC(-3.14159, 4)</code>	-3.1415
	<code>TRUNC(7419.917, 0)</code>	7419
	<code>TRUNC(7419.917, -1)</code>	7410
	<code>TRUNC(7419.917, -2)</code>	7400
	<code>TRUNC(7419.917, -4)</code>	0

**TRY** Attempts evaluation of expression and returns a default if the expression has a warning or error.

**Usage** `TRY(expression, default) -> value`

`expression` is the expression to be attempted. If the expression is evaluated without an error or warning, then the result of the expression is returned by `TRY`.

`default` is the another expression containing the default value if the evaluation of `expression` has a warning or error. The default expression must be of the same data type family as `expression`.

Examples	Expression	Result
	<code>TRY(NUM / ZERO, -1)</code>	-1
	<code>TRY(ZFLD + 0, \$NULL)</code>	ZFLD

The second example shows how to test a numeric field that may be invalid (such as an `IMAGE Z` field) for validity. If `ZFLD` is valid, adding 0 will return its value. If `ZFLD` is invalid, adding 0 will generate an error and the default value of `TRY` (in this case `$NULL`) is returned.

**TYPEOF** Returns a character string describing the data type of a variable or a record. For a record variable, the

word RECORD is returned.

Usage                    `TYPEOF(expression) -> string`

Examples                Assume the follow definitions:

```
DEFINE IMAGEI1VAR : IMAGE I1
DEFINE ODBCINTVAR: ODBC INTEGER
DEFINE ORACHAR20VAR: ORACLE CHAR(20)
DEFINE ORAREC : RECORD
    NUM : ORACLE CHAR(10)
    NAM : ORACLE CHAR(20)
    ADR : ORACLE CHAR(40)
END
```

Expression	Result
<code>TYPEOF(IMAGEI1VAR)</code>	"IMAGE I1"
<code>TYPEOF(ODBCINTVAR)</code>	"ODBC INTEGER"
<code>TYPEOF(ORACHAR20VAR)</code>	"ORACLE CHAR(20)"
<code>TYPEOF(ORAREC)</code>	"RECORD"

UDPRECV                Receives a datagram message from another system using UDP. UDP is a standard network data communications protocol similar to TCP, except that UDP messages are considered "unreliable". The intent of UDPRECV is to receive a "wake up" message from a Warehouse script running on another system.

Usage                    `UDPRECV(port, timeout) -> string`

`port` is the port number on which to listen for the message. Port numbers are standard on the internet and are assigned by The Internet Assigned Numbers Authority. See:

<http://www.iana.org/assignments/port-numbers>

This must be the same port to which the UDP message is sent and should be from 49152 through 65535.

`timeout` is the number of seconds to wait for a message. A timeout of 0 indicates to wait forever. If no message is received within timeout seconds,

then a zero length string is returned to UDPRECV.

#### Example

#### Statement

```
SETVAR ANS = UDPRECV( 54320 , 600 )
```

Listens for 10 minutes on port 54320 for a UDP message and sets the variable `ANS` to the message received.

#### UDPSEND

Sends a datagram message to another system using UDP. UDP is a standard network data communications protocol similar to TCP, except that UDP messages are considered "unreliable". The intent of `UDPSEND` is to send a "wake up" message to a Warehouse script running on another system. `UDPSEND` must be accessed with the `CALL` statement.

#### Usage

```
CALL UDPSEND(system, port, message)
```

`system` is the name or IP address of the system to which the message is to be sent.

`port` is the port number on the remote system to which the message is to be sent. Port numbers are standard on the internet and are assigned by The Internet Assigned Numbers Authority. See: <http://www.iana.org/assignments/port-numbers> This must be the same port to which the UDP message is sent and should be from 49152 through 65535.

`message` is a string of characters to be sent to the remote system.

#### Example

#### Statement

```
CALL UDPSEND( "UXSYS4" , 54320 , "WAKE" )
```

Sends the message `WAKE` to the remote system `UXSYS4` using port number 54320.

## UPS

Upshifts a string. UPS converts all lowercase characters in a given string to uppercase characters.

To downshift a string use the DWNS function.

## Usage

UPS(string) -> string

## Examples

<u>Expression</u>	<u>Result</u>
UPS("Taurus")	"TAURUS"
UPS("software")	"SOFTWARE"

## YYYYMMDD

Calculates a numeric date in YYYYMMDD format given a day number since Monday, December 31, 1900. The YYYYMMDD function accepts negative day numbers for dates prior to December 31, 1900.

YYYYMMDD is the inverse function of DAYNUM.

## Usage

YYYYMMDD(day-number) -> yyyyymmdd-number

## Examples

<u>Expression</u>	<u>Result</u>
YYYYMMDD(32327)	19890704
YYYYMMDD(-45469)	17760704
YYYYMMDD(1)	19010101

## **Chapter Six**

### **Data Types**

### Chapter Overview

This chapter describes in detail each of the data types supported by Warehouse. Every database supported by Warehouse comes with data types used by that database. For Warehouse to have the ability to manage your data, Warehouse must be able to operate on the data given to it by the database system. Warehouse provides the ability to operate on any data type it recognizes and to convert fields from one data type to another.



**Allbase Data Types**

The following data types originated on the HP 3000 database management system Allbase. Allbase is a "standard" SQL database management system, and hence the Allbase data types are all based on SQL data types.

Supported Allbase data types are as follows:

BINARY	Fixed length binary data
CHAR	Fixed length character data
DECIMAL	Fixed point numeric data
DOUBLE PRECISION	64 bit IEEE floating point data
FLOAT	IEEE floating point data
INTEGER	32 bit integer data
REAL	32 bit IEEE floating point data
SMALLINT	16 bit integer data
VARBINARY	Variable length binary data
VARCHAR	Variable length character data

**Null Values**

Allbase data types support null values. To indicate a data type that allows null values, `ALLOW NULLS` is appended to the data type specification.  
Example:

```
DEFINE CV : ALLBASE CHAR(20) ALLOW NULLS
```

The following is a detailed description of each of the Allbase data types supported by Warehouse.

**ALLBASE BINARY**

The `ALLBASE BINARY` data type represents fixed length binary data.

**Syntax**

```
ALLBASE BINARY(n)
```

`n` specifies the length in bytes of the field. `n` must be from 1 to 3996.

`ALLOW NULLS` may be appended to the data type

specification to allow storage of a null value.

#### Display

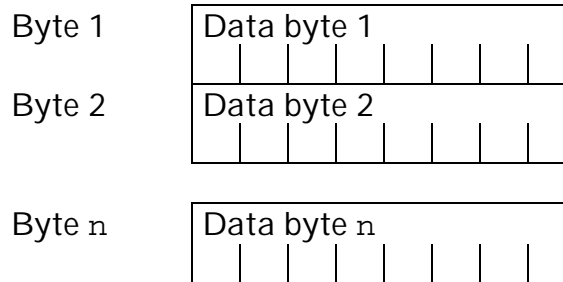
ALLBASE BINARY fields are printed in hexadecimal using a default print width of  $2 * (n + 1)$ .

#### Family

Binary, fixed length

#### Technical

The byte layout of ALLBASE BINARY items is as follows:



Size: n bytes

#### Examples

```
DEFINE BIN : ALLBASE BINARY(20)
DEFINE BIGBIN : ALLBASE BINARY(2000)
```

Defines BIN as a 20 byte binary field. Defines BIGBIN as a 2000 byte binary field.

#### ALLBASE CHAR

The ALLBASE CHAR data type represents fixed length character data.

#### Syntax

ALLBASE CHAR(n)

n specifies the length in bytes of the field. n must be from 1 to 3996.

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

#### Display

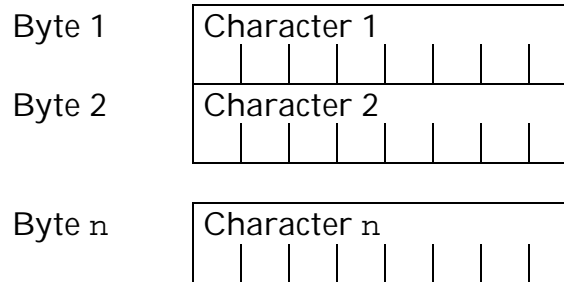
The default print width of ALLBASE CHAR fields is the width of the field, or n.

#### Family

Character string, fixed length

## Technical

The byte layout of ALLBASE CHAR items is as follows:



Size: n bytes

## Examples

```
DEFINE CH : ALLBASE CHAR(20)
DEFINE BIGCH : ALLBASE CHAR(2000)
```

Defines CH as a 20 byte fixed length character string. Defines BIGCH as a 2000 byte fixed length character string.

## ALLBASE DECIMAL

The ALLBASE DECIMAL data type represents fixed point numeric data.

## Syntax

```
DECIMAL(n)      or      DECIMAL(n,m)
DEC(n)          or      DEC(n,m)
NUMERIC(n)      or      NUMERIC(n,m)
```

The keywords DECIMAL, DEC, and NUMERIC have identical meanings and may be used interchangeably.

n specifies the maximum number of digits the field may hold, including digits to the right of the decimal point. n must be from 1 to 15.

If m is specified, m indicates the number of digits to the right of the decimal point the field may hold. If m is not specified, m is assumed to be 0. m must be from 0 to n.

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

Display                      The default print picture of ALLBASE DECIMAL fields is:

PIC "-(n-m)9.9(s)"      for m > 0  
 PIC "-(n)9"                for m = 0

Family                      Numeric

Technical                    The byte layout of an ALLBASE DECIMAL field depends on the size of n. If n is less than 8, ALLBASE DECIMAL fields require 4 bytes. If n is greater than or equal to 8, ALLBASE DECIMAL fields require 8 bytes.

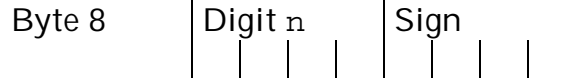
Byte layout for n from 1 to 7:

Byte 1	Digit n - 6 	Digit n - 5 
Byte 2	Digit n - 4 	Digit n - 3 
Byte 3	Digit n - 2 	Digit n - 1 
Byte 4	Digit n 	Sign 

Size: 4 bytes

Byte layout for n from 8 to 15:

Byte 1	Digit n - 14 	Digit n - 13 
Byte 2	Digit n - 12 	Digit n - 11 
Byte 3	Digit n - 10 	Digit n - 9 
Byte 4	Digit n - 8 	Digit n - 7 
Byte 5	Digit n - 6 	Digit n - 5 
Byte 6	Digit n - 4 	Digit n - 3 
Byte 7	Digit n - 2 	Digit n - 1 



Size: 8 bytes

Each digit of the number requires 4 bits and 4 bits is required for the sign.

Sign is defined as follows:

1111 = Unsigned

1100 = Positive

1101 = Negative

### Range

The range of ALLBASE DECIMAL items is:

Maximum: +99..99 with n - m digits

Minimum: -99..99 with n - m digits

### Examples

```
DEFINE DEC : ALLBASE DECIMAL(6)
DEFINE AMT : ALLBASE DEC(10,2)
```

Defines DEC as a 4 byte fixed decimal integer that can hold up to 6 digits. The default print picture for DEC is "-(6)9". Defines AMT as an 8 byte fixed decimal integer that can hold up to 10 digits: 8 digits to the left of the decimal point, and 2 digits to the right of the decimal point. The default print picture for AMT is "-(8)9.9(2)"

### ALLBASE DOUBLE PRECISION

The ALLBASE DOUBLE PRECISION data type represents an 8 byte IEEE floating point number.

### Syntax

ALLBASE DOUBLE PRECISION

or

ALLBASE FLOAT

or

ALLBASE FLOAT(n) where n >= 25 and n <= 53

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

### Display

The default print picture of ALLBASE DOUBLE PRECISION fields is:

PIC "-(9)9.9(6)"

Family

Numeric, floating point

Technical

The byte layout of the ALLBASE DOUBLE PRECISION data type is as follows:

Byte 1	s	e <sub>0</sub>	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>
Byte 2	e <sub>7</sub>	e <sub>8</sub>	e <sub>9</sub>	e <sub>10</sub>	f <sub>0</sub>	f <sub>1</sub>	f <sub>2</sub>	f <sub>3</sub>
Byte 3	f <sub>4</sub>	f <sub>5</sub>	f <sub>6</sub>	f <sub>7</sub>	f <sub>8</sub>	f <sub>9</sub>	f <sub>10</sub>	f <sub>11</sub>
Byte 4	f <sub>12</sub>	f <sub>13</sub>	f <sub>14</sub>	f <sub>15</sub>	f <sub>16</sub>	f <sub>17</sub>	f <sub>18</sub>	f <sub>19</sub>
Byte 5	f <sub>20</sub>	f <sub>21</sub>	f <sub>22</sub>	f <sub>23</sub>	f <sub>24</sub>	f <sub>25</sub>	f <sub>26</sub>	f <sub>27</sub>
Byte 6	f <sub>28</sub>	f <sub>29</sub>	f <sub>30</sub>	f <sub>31</sub>	f <sub>32</sub>	f <sub>33</sub>	f <sub>34</sub>	f <sub>35</sub>
Byte 7	f <sub>36</sub>	f <sub>37</sub>	f <sub>38</sub>	f <sub>39</sub>	f <sub>40</sub>	f <sub>41</sub>	f <sub>42</sub>	f <sub>43</sub>
Byte 8	f <sub>44</sub>	f <sub>45</sub>	f <sub>46</sub>	f <sub>47</sub>	f <sub>48</sub>	f <sub>49</sub>	f <sub>50</sub>	f <sub>51</sub>

Size: 8 bytes

s = sign bit, 1 = negative

e = exponent, 11 bits, biased by 1023

f = fraction, 52 bits, implicit leading 1

Value:  $(-1)^s * 2^{(e - 1023)} * (1.f)$

Notes: Exponents of all 0's and all 1's are reserved.

An exponent with all 0's and a fraction of all 0's represents the number 0. An exponent of all 0's and a non-zero fraction represents a denormalized number. An exponent of all 1's and a fraction of all 0's represents infinity. An exponent of all 1's and a non-zero fraction represents a NaN (not a number).

Range

The range of ALLBASE DOUBLE PRECISION items is:

Maximum:  $+1.797693134862318 \cdot 10^{308}$

Minimum:  $-1.797693134862318 \cdot 10^{308}$

Minimum > 0:  $4.940656458412465 \cdot 10^{-324}$

Examples

```
DEFINE FLT : ALLBASE DOUBLE PRECISION
DEFINE AMT : ALLBASE FLOAT(53)
```

Defines `FLT` as an IEEE 8 byte floating point number. Defines `AMT` as an IEEE 8 byte floating point number.

**ALLBASE FLOAT**

The `ALLBASE FLOAT` data type represents an IEEE floating point number.

**Syntax**

`ALLBASE FLOAT(n)`                      or `ALLBASE FLOAT`

`n` represents the number of bits of precision in the mantissa of the floating point number. If `n` is from 1 to 24, `n` is interpreted as 24 and the field is defined as a `REAL` field. If `n` is from 25 to 53, `n` is interpreted as 53 and the field is defined as a `ALLBASE DOUBLE PRECISION` field.

`n` must be from 1 to 53. If `n` is not specified, the default of 53 is used.

`ALLOW NULLS` may be appended to the data type specification to allow storage of a null value.

**Display**

The default print picture of `ALLBASE FLOAT` fields is:

```
PIC "-(9)9.9(6)"
```

**Family**

Numeric, floating point

**Technical**

The byte layout of the `ALLBASE FLOAT` data type depends on the size of `n`. If `n` is from 1 to 24, the field is interpreted as a `REAL` field; see the `ALLBASE REAL` section for details. If `n` is from 25 to 53, the field is interpreted as a `DOUBLE PRECISION` field; see the `ALLBASE DOUBLE PRECISION` section for details.

**Examples**

```
DEFINE RL   : ALLBASE FLOAT(24)
DEFINE FLT  : ALLBASE FLOAT(53)
DEFINE AMT  : ALLBASE FLOAT
```

Defines `RL` as an IEEE 4 byte floating point number. Defines `FLT` and `AMT` as IEEE 8 byte

floating point numbers.

## ALLBASE INTEGER

The ALLBASE INTEGER data type represents a 4 byte binary signed integer in twos-complement form.

### Syntax

ALLBASE INTEGER

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

### Display

The default print picture of ALLBASE INTEGER fields is:

PIC "-(10)9"

### Family

Numeric, binary integer

### Technical

The byte layout of the ALLBASE INTEGER data type is as follows:

Byte 1	s	b <sub>0</sub>	b <sub>1</sub>	b <sub>2</sub>	b <sub>3</sub>	b <sub>4</sub>	b <sub>5</sub>	b <sub>6</sub>
Byte 2	b <sub>7</sub>	b <sub>8</sub>	b <sub>9</sub>	b <sub>10</sub>	b <sub>11</sub>	b <sub>12</sub>	b <sub>13</sub>	b <sub>14</sub>
Byte 3	b <sub>15</sub>	b <sub>16</sub>	b <sub>17</sub>	b <sub>18</sub>	b <sub>19</sub>	b <sub>20</sub>	b <sub>21</sub>	b <sub>22</sub>
Byte 4	b <sub>23</sub>	b <sub>24</sub>	b <sub>25</sub>	b <sub>26</sub>	b <sub>27</sub>	b <sub>28</sub>	b <sub>29</sub>	b <sub>30</sub>

Size: 4 bytes

s = sign bit, 1 = negative

b<sub>n</sub> = b<sub>0</sub> is the most significant bit and b<sub>30</sub> is the least significant bit

### Range

The range of INTEGER items is:

Maximum: +2147483647

Minimum: -2147483648

### Examples

DEFINE INT : ALLBASE INTEGER

Defines INT as a 4 byte binary integer.

## ALLBASE REAL

The ALLBASE REAL data type represents a 4 byte IEEE floating point number.



## Syntax

ALLBASE REAL

or

ALLBASE FLOAT(*n*) where  $n \leq 24$ 

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

## Display

The default print picture of ALLBASE REAL fields is:

```
PIC "-(9)9.9(6)"
```

## Family

Numeric, floating point

## Technical

The byte layout of the ALLBASE REAL data type is as follows:

Byte 1	s	e <sub>0</sub>	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>
Byte 2	e <sub>7</sub>	f <sub>0</sub>	f <sub>1</sub>	f <sub>2</sub>	f <sub>3</sub>	f <sub>4</sub>	f <sub>5</sub>	f <sub>6</sub>
Byte 3	f <sub>7</sub>	f <sub>8</sub>	f <sub>9</sub>	f <sub>10</sub>	f <sub>11</sub>	f <sub>12</sub>	f <sub>13</sub>	f <sub>14</sub>
Byte 4	f <sub>15</sub>	f <sub>16</sub>	f <sub>17</sub>	f <sub>18</sub>	f <sub>19</sub>	f <sub>20</sub>	f <sub>21</sub>	f <sub>22</sub>

Size: 4 bytes

s = sign bit, 1 = negative

e = exponent, 8 bits, biased by 127

f = fraction, 23 bits, implicit leading 1

Value:  $(-1)^s * 2^{(e - 127)} * (1.f)$

Notes: Exponents of all 0's and all 1's are reserved.

An exponent with all 0's and a fraction of all 0's represents the number 0. An exponent of all 0's and a non-zero fraction represents a denormalized number. An exponent of all 1's and a fraction of all 0's represents infinity. An exponent of all 1's and a non-zero fraction represents a NaN (not a number).

## Range

The range of ALLBASE REAL items is:

Maximum:  $+3.402823 \cdot 10^{38}$ Minimum:  $-3.402823 \cdot 10^{38}$ Minimum > 0:  $+1.175495 \cdot 10^{-38}$

## Examples

```
DEFINE FLT : ALLBASE REAL
DEFINE AMT : ALLBASE FLOAT( 24 )
```

Defines FLT as an IEEE 4 byte floating point number. Defines AMT as an IEEE 4 byte floating point number.

## ALLBASE SMALLINT

The ALLBASE SMALLINT data type represents a 2 byte binary signed integer in twos-complement form.

## Syntax

```
ALLBASE SMALLINT
```

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

## Display

The default print picture of ALLBASE SMALLINT fields is:

```
PIC "-(5)9"
```

## Family

Numeric, binary integer

## Technical

The byte layout of the ALLBASE SMALLINT data type is as follows:

Byte 1	s	b <sub>0</sub>	b <sub>1</sub>	b <sub>2</sub>	b <sub>3</sub>	b <sub>4</sub>	b <sub>5</sub>	b <sub>6</sub>
Byte 2	b <sub>7</sub>	b <sub>8</sub>	b <sub>9</sub>	b <sub>10</sub>	b <sub>11</sub>	b <sub>12</sub>	b <sub>13</sub>	b <sub>14</sub>

Size: 2 bytes

s = sign bit, 1 = negative

b<sub>n</sub>=b<sub>0</sub> is the most significant bit and b<sub>14</sub> is the least significant bit.

## Range

The range of ALLBASE SMALLINT items is:

Maximum: +32767

Minimum: -32768

## Examples

```
DEFINE INT : ALLBASE SMALLINT
```

Defines INT as a 2 byte binary integer.

ALLBASE VARBINARY      The ALLBASE VARBINARY data type represents variable length binary data.

Syntax      ALLBASE VARBINARY(n)

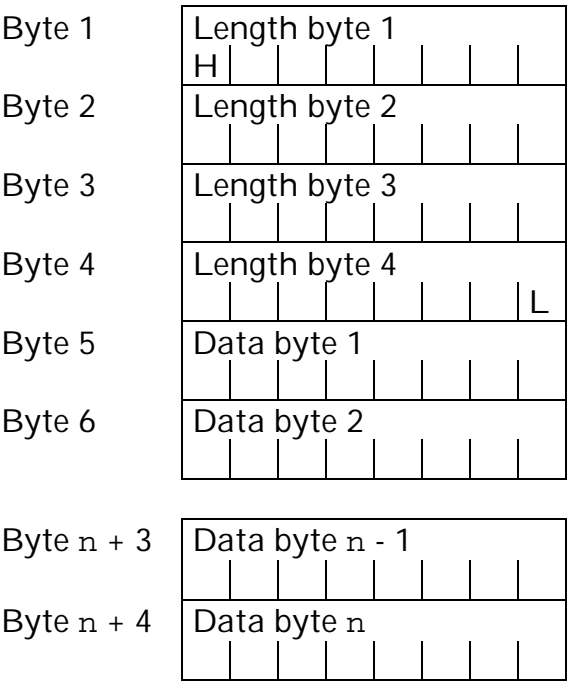
n specifies the maximum length in bytes of the field. n must be from 1 to 3996.

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

Display      ALLBASE VARBINARY fields are printed in hexadecimal using a default print width of  $2 * (n + 1)$ .

Family      Binary, variable length

Technical      The byte layout of ALLBASE VARBINARY items is as follows:



Size: n + 4 bytes  
H=    High order bit  
L=    Low order bit

## Examples

```
DEFINE BIN : ALLBASE VARBINARY(20)
DEFINE BIGBIN : ALLBASE VARBINARY(2000)
```

Defines BIN as a variable length binary string capable of holding up to 20 bytes. Defines BIGBIN as a variable length binary string capable of holding up to 2000 bytes.

## ALLBASE VARCHAR

The ALLBASE VARCHAR data type represents variable length character data.

## Syntax

```
ALLBASE VARCHAR(n)
```

n specifies the maximum length in bytes of the field. n must be from 1 to 3996.

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

## Display

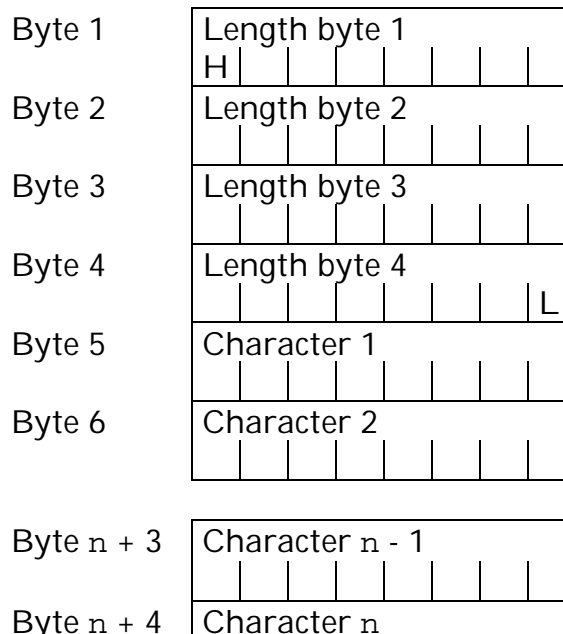
The default print width of ALLBASE VARCHAR fields is the width of the field, or n.

## Family

Character string, variable length

## Technical

The byte layout of ALLBASE VARCHAR items is as follows:





Size:  $n + 4$  bytes

H= High order bit

L= Low order bit

### Examples

```
DEFINE CH : ALLBASE VARCHAR(20)
DEFINE BIGCH : ALLBASE VARCHAR(2000)
```

Defines CH as a variable length character string capable of holding up to 20 characters. Defines BIGCH as a variable length character string capable of holding up to 2000 characters.

**IMAGE Data Types**      The following data types originated on the HP 3000 database management system IMAGE.

IMAGE data types are specified as follows:

```
IMAGE [count] type length  
  
or more simply  
  
[count] type length
```

Where `count` is the number of the times the item repeats, as in an array. The default `count` is 1. The `type` is the item type and must be one of the types listed below. The `length` is the length of the item and its exact meaning depends upon `type`.

Supported IMAGE data types are:

E	IEEE floating point data
I	Binary integer data
J	Binary integer data
K	Unsigned binary integer data
P	Packed decimal data
R	HP3000 floating point data
U	Uppercase character data
X	Character data
Z	Zoned decimal data

**Null Values**      Image data types do *not* support null values.

The following is a detailed description of each of the IMAGE data types supported by Warehouse.

IMAGE E2      The IMAGE E2 data type represents a 4 byte IEEE  
IMAGE\_ E2      floating point number. IMAGE\_ uses little-endian  
storage.

Syntax	IMAGE E2	or	E2
	IMAGE kE2	or	kE2

Where `k` represents the number of items. When `k`

is greater than 1, an array of `k IMAGE E2` items is specified.

## Display

The default print picture of `IMAGE E2` fields is:

```
PIC "-(9)9.9(6)"
```

## Family

Numeric, floating point

## Technical

The byte layout of the `IMAGE E2` data type is as follows:

Byte 1	s	e <sub>0</sub>	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>
Byte 2	e <sub>7</sub>	f <sub>0</sub>	f <sub>1</sub>	f <sub>2</sub>	f <sub>3</sub>	f <sub>4</sub>	f <sub>5</sub>	f <sub>6</sub>
Byte 3	f <sub>7</sub>	f <sub>8</sub>	f <sub>9</sub>	f <sub>10</sub>	f <sub>11</sub>	f <sub>12</sub>	f <sub>13</sub>	f <sub>14</sub>
Byte 4	f <sub>15</sub>	f <sub>16</sub>	f <sub>17</sub>	f <sub>18</sub>	f <sub>19</sub>	f <sub>20</sub>	f <sub>21</sub>	f <sub>22</sub>

Size: 4 bytes

s = sign bit, 1 = negative

e = exponent, 8 bits, biased by 127

f = fraction, 23 bits, implicit leading 1

Value:  $(-1)^s * 2^{(e - 127)} * (1.f)$

Notes: Exponents of all 0's and all 1's are reserved.

An exponent with all 0's and a fraction of all 0's represents the number 0. An exponent of all 0's and a non-zero fraction represents a denormalized number. An exponent of all 1's and a fraction of all 0's represents infinity. An exponent of all 1's and a non-zero fraction represents a NaN (not a number).

## Range

The range of `IMAGE E2` items is:

Maximum:  $+3.402823 \cdot 10^{38}$

Minimum:  $-3.402823 \cdot 10^{38}$

Minimum > 0:  $+1.175495 \cdot 10^{-38}$

## Examples

```
DEFINE FLT : E2
```

```
DEFINE FARRAY : IMAGE 12E2
```

Defines `FLT` as an IEEE 4 byte floating point number. Defines `FARRAY` as an array of 12 `IMAGE`

E2 numbers.

IMAGE E4  
IMAGE\_ E4

The IMAGE E4 data type represents an 8 byte IEEE floating point number. IMAGE\_ uses little-endian storage.

Syntax

IMAGE E4                      or                      E4  
IMAGE kE4                    or                      kE4

Where k represents the number of items. When k is greater than 1, an array of k IMAGE E4 items is specified.

Display

The default print picture of IMAGE E4 fields is:

PIC "-(9)9.9(6)"

Family

Numeric, floating point

Technical

The byte layout of the IMAGE E4 data type is as follows:

Byte 1	s	e <sub>0</sub>	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>
Byte 2	e <sub>7</sub>	e <sub>8</sub>	e <sub>9</sub>	e <sub>10</sub>	f <sub>0</sub>	f <sub>1</sub>	f <sub>2</sub>	f <sub>3</sub>
Byte 3	f <sub>4</sub>	f <sub>5</sub>	f <sub>6</sub>	f <sub>7</sub>	f <sub>8</sub>	f <sub>9</sub>	f <sub>10</sub>	f <sub>11</sub>
Byte 4	f <sub>12</sub>	f <sub>13</sub>	f <sub>14</sub>	f <sub>15</sub>	f <sub>16</sub>	f <sub>17</sub>	f <sub>18</sub>	f <sub>19</sub>
Byte 5	f <sub>20</sub>	f <sub>21</sub>	f <sub>22</sub>	f <sub>23</sub>	f <sub>24</sub>	f <sub>25</sub>	f <sub>26</sub>	f <sub>27</sub>
Byte 6	f <sub>28</sub>	f <sub>29</sub>	f <sub>30</sub>	f <sub>31</sub>	f <sub>32</sub>	f <sub>33</sub>	f <sub>34</sub>	f <sub>35</sub>
Byte 7	f <sub>36</sub>	f <sub>37</sub>	f <sub>38</sub>	f <sub>39</sub>	f <sub>40</sub>	f <sub>41</sub>	f <sub>42</sub>	f <sub>43</sub>
Byte 8	f <sub>44</sub>	f <sub>45</sub>	f <sub>46</sub>	f <sub>47</sub>	f <sub>48</sub>	f <sub>49</sub>	f <sub>50</sub>	f <sub>51</sub>

Size: 8 bytes

s = sign bit, 1 = negative

e = exponent, 11 bits, biased by 1023

f = fraction, 52 bits, implicit leading 1

Value:  $(-1)^s * 2^{(e - 1023)} * (1.f)$

Notes: Exponents of all 0's and all 1's are reserved.  
An exponent with all 0's and a fraction of all 0's represents the number 0. An exponent of all 0's and a non-zero fraction represents a



denormalized number. An exponent of all 1's and a fraction of all 0's represents infinity. An exponent of all 1's and a non-zero fraction represents a NaN (not a number).

## Range

The range of `E4` items is:

Maximum:  $+1.797693134862318 \cdot 10^{308}$   
 Minimum:  $-1.797693134862318 \cdot 10^{308}$   
 Minimum > 0:  $4.940656458412465 \cdot 10^{-324}$

## Examples

```
DEFINE FLT : E4
DEFINE FARRAY : 12E4
```

Defines `FLT` as an IEEE 8 byte floating point number. Defines `FARRAY` as an array of 12 `IMAGE E4` numbers.

```
IMAGE I1
IMAGE J1
IMAGE_ I1
IMAGE_ J1
```

The `IMAGE I1` data type and the `IMAGE J1` data type are considered identical by Warehouse and represent a 2 byte binary signed integer in twos-complement form. `IMAGE_` uses little-endian storage.

## Syntax

```
IMAGE I1      or      I1
IMAGE kI1     or      kI1
IMAGE J1      or      J1
IMAGE kJ1     or      kJ1
```

Where `k` represents the number of items. When `k` is greater than 1, an array of `k IMAGE I1/J1` items is specified.

## Display

The default print picture of `IMAGE I1` fields is:

```
PIC "(5)9"
```

## Family

Numeric, binary integer

## Technical

The byte layout of the `IMAGE I1` and `IMAGE J1` data types are as follows:

Byte 1     

s	H						
---	---	--	--	--	--	--	--

Byte 2 

							L
--	--	--	--	--	--	--	---

Size: 2 bytes

s = sign bit, 1 = negative

H= High order bit

L= Low order bit

Range

The range of IMAGE I1 and IMAGE J1 items is:

Maximum: +32767

Minimum: -32768

Examples

```
DEFINE INT : IMAGE I1
DEFINE IARRAY : IMAGE 12J1
```

Defines INT as a 2 byte binary integer. Defines IARRAY as an array of 12 J1 numbers.

```
IMAGE I2
IMAGE J2
IMAGE_ I2
IMAGE_ J2
```

The IMAGE I2 data type and the IMAGE J2 data type are considered identical by Warehouse and represent a 4 byte binary signed integer in twos-complement form. IMAGE\_ uses little-endian storage.

Syntax

```
IMAGE I2      or      I2
IMAGE kI2     or      kI2
IMAGE J2      or      J2
IMAGE kJ2     or      kJ2
```

Where k represents the number of items. When k is greater than 1, an array of k IMAGE I2/J2 items is specified.

Display

The default print picture of IMAGE I2 and IMAGE J2 fields is:

```
PIC "-(10)9"
```

Family

Numeric, binary integer

Technical

The byte layout of the IMAGE I2 and IMAGE J2 data types are as follows:

Byte 1	s	H					
Byte 2							
Byte 3							
Byte 4							L

Size: 4 bytes

s = sign bit, 1 = negative

H= High order bit

L= Low order bit

### Range

The range of IMAGE I2 and IMAGE J2 items is:

Maximum: +2147483647

Minimum: -2147483648

### Examples

```
DEFINE INT : IMAGE I2
DEFINE IARRAY : IMAGE 12J2
```

Defines INT as a 4 byte binary integer. Defines IARRAY as an array of 12 IMAGE J2 numbers.

IMAGE In  
IMAGE Jn  
IMAGE Kn

The data types IMAGE In, IMAGE Jn, and IMAGE Kn data types where n is from 3 to 8 are considered identical by Warehouse and represent binary signed integers in twos-complement form.

### Syntax

IMAGE In	or	In
IMAGE kIn	or	kIn
IMAGE Jn	or	Jn
IMAGE kJn	or	kJn
IMAGE Kn	or	Kn
IMAGE kKn	or	kKn

Where n represents the number of 16 bit words and k represents the number of items. The value of n must be from 1 to 8. When k is greater than 1, an array of k IMAGE In/Jn items is specified.

### Display

The default print picture of IMAGE In, IMAGE Jn and IMAGE Kn fields is:

```

IMAGE I3:      PIC "-(15)9"
IMAGE I4:      PIC "-(19)9"
IMAGE I5:      PIC "-(24)9"
IMAGE I6:      PIC "-(29)9"
IMAGE I7:      PIC "-(34)9"
IMAGE I8:      PIC "-(39)9"

```

Family                      Numeric, binary integer

Technical                      The byte layout of the IMAGE In, IMAGE Jn and IMAGE Kn data types are as follows:

Byte 1	s	H					
Byte 2							

Byte 2n-1							
Byte 2n							L

Size: 2n bytes

s =     sign bit, 1 = negative  
H=     High order bit  
L=     Low order bit

Range                          The range of IMAGE In, IMAGE Jn and IMAGE Kn items is:

Maximum values are:

```

I3:      +140737488355327
I4:      +9223372036854775807
I5:      +604462909807314587353087
I6:      +39614081257132168796771975167
I7:      +2596148429267413814265248164610047
I8: +170141183460469231731687303715884105727

```

Minimum values are:

```

I3:      -140737488355328
I4:      -9223372036854775808
I5:      -604462909807314587353088
I6:      -39614081257132168796771975168
I7:      -2596148429267413814265248164610048
I8: -170141183460469231731687303715884105728

```

Examples                      `DEFINE INT : IMAGE I4`  
                                 `DEFINE IARRAY : IMAGE 12J4`

Defines INT as a 8 byte binary integer. Defines IARRAY as an array of 12 IMAGE J4 numbers.

IMAGE K1  
IMAGE\_ K1

The IMAGE K1 data type represents a 2 byte binary *unsigned* integer in twos-complement form. IMAGE\_ uses little-endian storage.

Syntax

IMAGE K1                    or                    K1  
IMAGE kK1                  or                    kK1

Where k represents the number of items. When k is greater than 1, an array of k IMAGE K1 items is specified.

Display

The default print picture of IMAGE K1 fields is:

PIC "Z(5)9"

Family

Numeric, binary integer

Technical

The byte layout of the IMAGE K1 data type is as follows:

Byte 1	H							
Byte 2								L

Size: 2 bytes

H= High order bit

L= Low order bit

Range

The range of IMAGE K1 items is:

Maximum: +65535

Minimum: 0

Examples

```
DEFINE INT : IMAGE K1
DEFINE IARRAY : IMAGE 12K1
```

Defines INT as a 2 byte unsigned binary integer. Defines IARRAY as an array of 12 IMAGE K1 numbers.

IMAGE K2  
IMAGE\_ K2

The IMAGE K2 data type represents a 4 byte binary *unsigned* integer in twos-complement form. IMAGE\_ uses little-endian storage.

Syntax

IMAGE K2                    or                    K2  
IMAGE kK2                  or                    kK2

Where k represents the number of items. When k is greater than 1, an array of k IMAGE K2 items is specified.

Display

The default print picture of IMAGE K2 fields is:

PIC "Z(10)9"

Family

Numeric, binary integer

Technical

The byte layout of the IMAGE K2 data type is as follows:

Byte 1	H						
Byte 2							
Byte 3							
Byte 4							L

Size: 4 bytes

H=    High order bit

L=    Low order bit

Range

The range of IMAGE K2 items is:

Maximum:        +4294967295

Minimum:        0

Examples

```
DEFINE INT : IMAGE K2
DEFINE IARRAY : IMAGE 12K2
```

Defines INT as a 4 byte unsigned binary integer.  
Defines IARRAY as an array of 12 IMAGE K2 numbers.

**IMAGE P** The **IMAGE P** data type is used to represent fixed length packed decimal integers with no decimal point.

**Syntax** `IMAGE Pn` or `Pn`  
`IMAGE kPn` or `kPn`

Where *n* is an even number that represents one more than the number of digits the field can represent. For example, an **IMAGE P8** field can represent 7 digits and a sign.

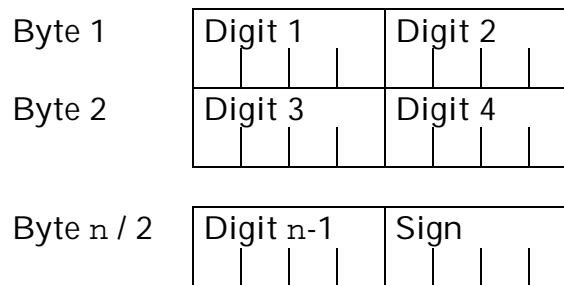
Where *k* represents the number of items. When *k* is greater than 1, an array of *k* **IMAGE P<sub>n</sub>** items is specified.

**Display** The default print picture of **IMAGE P** fields is:

`PIC "(n-1)9"`

**Family** Numeric, packed decimal integer

**Technical** The byte layout of an **IMAGE P** data type is as follows:



Size: *n* / 2 bytes

Each digit of the number requires 4 bits and 4 bits is required for the sign.

Sign is defined as follows:

1111 = Unsigned (hexadecimal F)

1100 = Positive (hexadecimal C)

1101 = Negative (hexadecimal D)

**Range** The range of **IMAGE P<sub>n</sub>** items is:

Maximum: +99..99 with n-1 digits

Minimum: -99..99 with n-1 digits

### Examples

```
DEFINE PCK : P12
DEFINE PARRAY : IMAGE 12P4
```

Defines PCK as a 6 byte packed decimal integer capable of holding up to 11 digit numbers. Defines PARRAY as an array of 12 IMAGE P4 numbers, each capable of holding up to 3 digits.

IMAGE R2  
IMAGE\_ R2

The IMAGE R2 data type represents a 4 byte HP3000 floating point number. IMAGE\_ uses little-endian storage.

### Syntax

```
IMAGE R2          or          R2
IMAGE kR2         or         kR2
```

Where k represents the number of items. When k is greater than 1, an array of k IMAGE R2 items is specified.

### Display

The default print picture of IMAGE R2 fields is:

```
PIC "-(9)9.9(6)"
```

### Family

Numeric, floating point

### Technical

The byte layout of the IMAGE R2 data type is as follows:

Byte 1	s	e <sub>0</sub>	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>
Byte 2	e <sub>7</sub>	e <sub>8</sub>	f <sub>0</sub>	f <sub>1</sub>	f <sub>2</sub>	f <sub>3</sub>	f <sub>4</sub>	f <sub>5</sub>
Byte 3	f <sub>6</sub>	f <sub>7</sub>	f <sub>8</sub>	f <sub>9</sub>	f <sub>10</sub>	f <sub>11</sub>	f <sub>12</sub>	f <sub>13</sub>
Byte 4	f <sub>14</sub>	f <sub>15</sub>	f <sub>16</sub>	f <sub>17</sub>	f <sub>18</sub>	f <sub>19</sub>	f <sub>20</sub>	f <sub>21</sub>

Size: 4 bytes

s = sign bit, 1 = negative

e = exponent, 9 bits, biased by 256

f = fraction, 22 bits, implicit leading 1

Value:  $(-1)^s * 2^{(e - 256)} * (1.f)$



Notes: All bits 0 represents 0.

## Range

The range of IMAGE R2 items is:

Maximum:  $+1.157920 \cdot 10^{77}$

Minimum:  $-1.157920 \cdot 10^{77}$

Minimum > 0:  $+8.636169 \cdot 10^{-78}$

## Examples

```
DEFINE FLT : R2
```

```
DEFINE FARRAY : IMAGE 12R2
```

Defines FLT as an HP3000 4 byte floating point number. Defines FARRAY as an array of 12 IMAGE R2 numbers.

IMAGE R4  
IMAGE\_ R4

The IMAGE R4 data type represents an 8 byte HP3000 floating point number. IMAGE\_ uses little-endian storage.

## Syntax

```
IMAGE R4          or          R4
IMAGE kR4         or          kR4
```

Where k represents the number of items. When k is greater than 1, an array of k IMAGE R4 items is specified.

## Display

The default print picture of IMAGE R4 fields is:

```
PIC "-(9)9.9(6)"
```

## Family

Numeric, floating point

## Technical

The byte layout of the IMAGE R4 data type is as follows:

Byte 1	s	e <sub>0</sub>	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>
Byte 2	e <sub>7</sub>	e <sub>8</sub>	f <sub>0</sub>	f <sub>1</sub>	f <sub>2</sub>	f <sub>3</sub>	f <sub>4</sub>	f <sub>5</sub>
Byte 3	f <sub>6</sub>	f <sub>7</sub>	f <sub>8</sub>	f <sub>9</sub>	f <sub>10</sub>	f <sub>11</sub>	f <sub>12</sub>	f <sub>13</sub>
Byte 4	f <sub>14</sub>	f <sub>15</sub>	f <sub>16</sub>	f <sub>17</sub>	f <sub>18</sub>	f <sub>19</sub>	f <sub>20</sub>	f <sub>21</sub>
Byte 5	f <sub>22</sub>	f <sub>23</sub>	f <sub>24</sub>	f <sub>25</sub>	f <sub>26</sub>	f <sub>27</sub>	f <sub>28</sub>	f <sub>29</sub>
Byte 6	f <sub>30</sub>	f <sub>31</sub>	f <sub>32</sub>	f <sub>33</sub>	f <sub>34</sub>	f <sub>35</sub>	f <sub>36</sub>	f <sub>37</sub>

Byte 7	$f_{38}$	$f_{39}$	$f_{40}$	$f_{41}$	$f_{42}$	$f_{43}$	$f_{44}$	$f_{45}$
Byte 8	$f_{46}$	$f_{47}$	$f_{48}$	$f_{49}$	$f_{50}$	$f_{51}$	$f_{52}$	$f_{53}$

Size: 8 bytes

s = sign bit, 1 = negative

e = exponent, 9 bits, biased by 256

f = fraction, 54 bits, implicit leading 1

Value:  $(-1)^s * 2^{(e - 256)} * (1.f)$

Notes: All bits 0 represents 0.

Range

The range of IMAGE R4 items is:

Maximum:  $+1.157920892373161 \cdot 10^{77}$

Minimum:  $-1.157920892373161 \cdot 10^{77}$

Minimum > 0:  $+8.636168555094445 \cdot 10^{-78}$

Examples

```
DEFINE FLT : R4
DEFINE FARRAY : IMAGE 12R4
```

Defines FLT as an HP3000 8 byte floating point number. Defines FARRAY as an array of 12 R4 IMAGE numbers.

IMAGE U

The IMAGE U data type is used to represent fixed length uppercase character data. Data stored into a U field is first converted to uppercase by Warehouse.

Syntax

```
IMAGE Un      or      Un
IMAGE kUn     or      kUn
```

Where n represents the number of characters the field can represent.

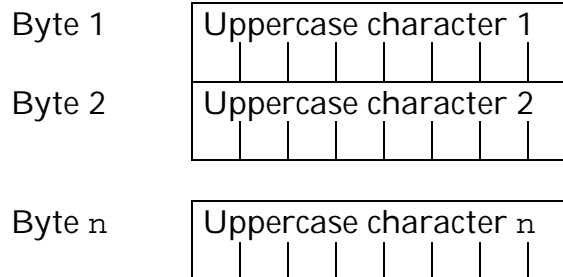
Where k represents the number of items. When k is greater than 1, an array of k IMAGE Un items is specified.

Display

The default print width of IMAGE U fields is the width of the field, or n.

Family Character string, fixed length

Technical The byte layout of `IMAGE U` type items is as follows:



Size:  $n$  bytes  
Each character requires 1 byte.

Examples

```
DEFINE CHAR : U12
DEFINE UARRAY : IMAGE 12U4
```

Defines `CHAR` as a 12 character field of uppercase characters. Defines `UARRAY` as an array of 12 `U4` `IMAGE` fields.

`IMAGE X` The `IMAGE X` data type is used to represent fixed length character data.

Syntax

<code>IMAGE Xn</code>	or	<code>Xn</code>
<code>IMAGE kXn</code>	or	<code>kXn</code>

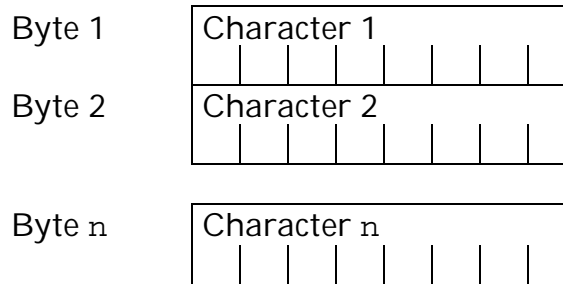
Where  $n$  represents the number of characters the field can represent.

Where  $k$  represents the number of items. When  $k$  is greater than 1, an array of  $k$  `IMAGE Xn` items is specified.

Display The default print width of `IMAGE X` fields is the width of the field, or  $n$ .

Family Character string, fixed length

Technical The byte layout of `IMAGE X` type items is as follows:



Size: n bytes

Each character requires 1 byte.

### Examples

```
DEFINE CHAR : X12
DEFINE XARRAY : IMAGE 12X4
```

Defines CHAR as 12 character field. Defines XARRAY as an array of 12 IMAGE X4 fields.

### IMAGE Z

The IMAGE Z data type is used to represent fixed length zoned decimal integers with no decimal point.

### Syntax

```
IMAGE Zn      or      Zn
IMAGE kZn     or      kZn
```

where n represents the number of digits the field can represent. For example, an IMAGE Z8 field can represent 8 digits and a sign.

Where k represents the number of items. When k is greater than 1, an array of k IMAGE Zn items is specified.

### Display

The default print picture of IMAGE Z fields is:

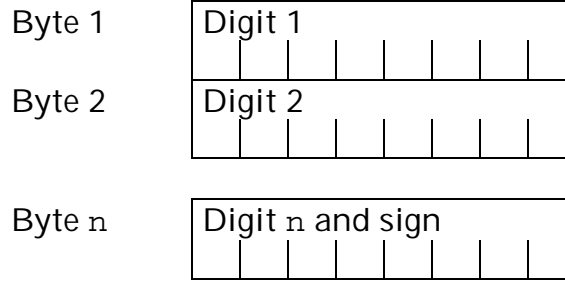
```
PIC "(n)9"
```

### Family

Numeric, zoned decimal integer

### Technical

The byte layout of IMAGE Z type items is as follows:



Size: n bytes

Each digit of the number requires 1 byte with the last byte containing both the last digit of the number and the sign. The last byte indicates the sign and last digit as follows:

Byte n	Unsigned	Positive	Negative
0	0	{	}
1	1	A	J
2	2	B	K
3	3	C	L
4	4	D	M
5	5	E	N
6	6	F	O
7	7	G	P
8	8	H	Q
9	9	I	R

Range

The range of IMAGE Z<sub>n</sub> items is:

Maximum: +99..99 with n digits

Minimum: -99..99 with n digits

Examples

```
DEFINE ZON : Z12
DEFINE ZARRAY : IMAGE 12Z4
```

Defines ZON as a zoned decimal integer capable of holding up to 12 digits. Defines ZARRAY as an array of 12 IMAGE Z4 numbers, each capable of holding up to 4 digits.

**ODBC Data Types**

The following data types are based on the Open Database Connectivity standard from Microsoft Corporation.

Supported ODBC data types are as follows:

BIGINT	64 bit integer data
BINARY	Fixed length binary data
BIT	Logical (True/False) data
CHAR	Fixed length character data
DATE	Calendar date
DECIMAL	Fixed point numeric data
DOUBLE PRECISION	64 bit IEEE floating point data
INTEGER	32 bit integer data
LONG VARBINARY	Long variable length binary data
LONG VARCHAR	Long variable length character data
NUMERIC	Alias for DECIMAL
REAL	32 bit IEEE floating point data
SMALLINT	16 bit integer data
TIME	Time of day
TIMESTAMP	Calendar date and time
TINYINT	8 bit integer data
UNIQUEIDENTIFIER	16 byte unique ID.
VARBINARY	Variable length binary data
VARCHAR	Variable length character data

**Null Values**

ODBC data types support null values. To indicate a data type that allows null values, `ALLOW NULLS` is appended to the data type specification.  
Example:

```
DEFINE CV : ODBC CHAR(20) ALLOW NULLS
```

The following is a detailed description of each of the ODBC data types supported by Warehouse.

## ODBC BIGINT

The ODBC BIGINT data type represents an 8 byte binary signed integer in twos-complement form.

## Syntax

ODBC BIGINT

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

## Display

The default print picture of ODBC BIGINT fields is:

PIC "-(19)9"

## Family

Numeric, binary integer

## Technical

The byte layout of the ODBC INTEGER data type is as follows:

Byte 1	b <sub>55</sub>	b <sub>56</sub>	b <sub>57</sub>	b <sub>58</sub>	b <sub>59</sub>	b <sub>60</sub>	b <sub>61</sub>	b <sub>62</sub>
Byte 2	b <sub>47</sub>	b <sub>48</sub>	b <sub>49</sub>	b <sub>50</sub>	b <sub>51</sub>	b <sub>52</sub>	b <sub>53</sub>	b <sub>54</sub>
Byte 3	b <sub>39</sub>	b <sub>40</sub>	b <sub>41</sub>	b <sub>42</sub>	b <sub>43</sub>	b <sub>44</sub>	b <sub>45</sub>	b <sub>46</sub>
Byte 4	b <sub>31</sub>	b <sub>32</sub>	b <sub>33</sub>	b <sub>34</sub>	b <sub>35</sub>	b <sub>36</sub>	b <sub>37</sub>	b <sub>38</sub>
Byte 5	b <sub>23</sub>	b <sub>24</sub>	b <sub>25</sub>	b <sub>26</sub>	b <sub>27</sub>	b <sub>28</sub>	b <sub>29</sub>	b <sub>30</sub>
Byte 6	b <sub>15</sub>	b <sub>16</sub>	b <sub>17</sub>	b <sub>18</sub>	b <sub>19</sub>	b <sub>20</sub>	b <sub>21</sub>	b <sub>22</sub>
Byte 7	b <sub>7</sub>	b <sub>8</sub>	b <sub>9</sub>	b <sub>10</sub>	b <sub>11</sub>	b <sub>12</sub>	b <sub>13</sub>	b <sub>14</sub>
Byte 8	b <sub>0</sub>	b <sub>1</sub>	b <sub>2</sub>	b <sub>3</sub>	b <sub>4</sub>	b <sub>5</sub>	b <sub>6</sub>	

Size: 4 bytes

s = sign bit, 1 = negative

b<sub>n</sub> = b<sub>0</sub> is the most significant bit and b<sub>62</sub> is the least significant bit

## Range

The range of INTEGER items is:

Maximum: +9223372036854775807

Minimum: -9223372036854775808

## Examples

DEFINE INT : ODBC BIGINT

Defines INT as an 8 byte binary integer.

## ODBC BINARY

The ODBC BINARY data type represents fixed length binary data.

**Syntax** ODBC BINARY(*n*)

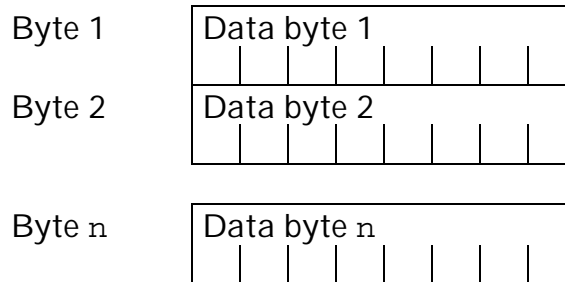
*n* specifies the length in bytes of the field. *n* must be from 1 to 8000.

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

**Display** ODBC BINARY fields are printed in hexadecimal using a default print width of 2 \* (*n* + 1).

**Family** Binary, fixed length

**Technical** The byte layout of ODBC BINARY items is as follows:



Size: *n* bytes

**Examples** DEFINE BIN : ODBC BINARY(20)  
 DEFINE BIGBIN : ODBC BINARY(2000)

Defines BIN as a 20 byte binary field. Defines BIGBIN as a 2000 byte binary field.

ODBC BIT The ODBC BIT data type represents Boolean, or true false data.

**Syntax** ODBC BIT

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

**Display** The default print width of BIT fields is 6.

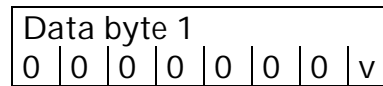
**Family** Logical



## Technical

The byte layout of ODBC `BINARY` items is as follows:

Byte 1



Size: 1 byte

## Range

An ODBC `BINARY` item may only represent `$TRUE` or `$FALSE`.

## Example

```
DEFINE MYFLAG : ODBC BINARY
SETVAR MYFLAG = $TRUE
```

Defines `MYFLAG` as an ODBC bit field, then sets the value of `MYFLAG` to `$TRUE`.

ODBC `CHAR`

The ODBC `CHAR` data type represents fixed length character data.

## Syntax

ODBC `CHAR`(*n*)

*n* specifies the length in bytes of the field. *n* must be from 1 to 8000.

`ALLOW NULLS` may be appended to the data type specification to allow storage of a null value.

## Display

The default print width of ODBC `CHAR` fields is the width of the field, or *n*.

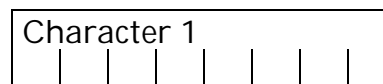
## Family

Character string, fixed length

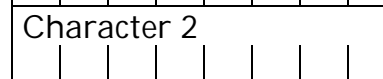
## Technical

The byte layout of ODBC `CHAR` items is as follows:

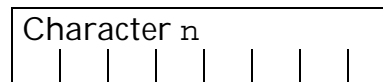
Byte 1



Byte 2



Byte *n*



Size: n bytes

## Examples

```
DEFINE CH : ODBC CHAR(20) ALLOW NULLS
DEFINE BIGCH : ODBC CHAR(200)
```

Defines CH as a 20 byte fixed length character string that allows null values. Defines BIGCH as a 200 byte fixed length character string.

## ODBC DATE

The ODBC DATE data type represents a date and/or time.

## Syntax

ODBC DATE

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

## Display

The default print width of ODBC DATE fields is 11 in the following format:  
dd-mmm-yyyy

## Family

Date/time

## Technical

The byte layout of ODBC DATE items is as follows:

Byte 1	Year low bits
Byte 2	Year high bits
Byte 3	Month
Byte 4	Month filler (zero)
Byte 5	Day
Byte 6	Day filler (zero)

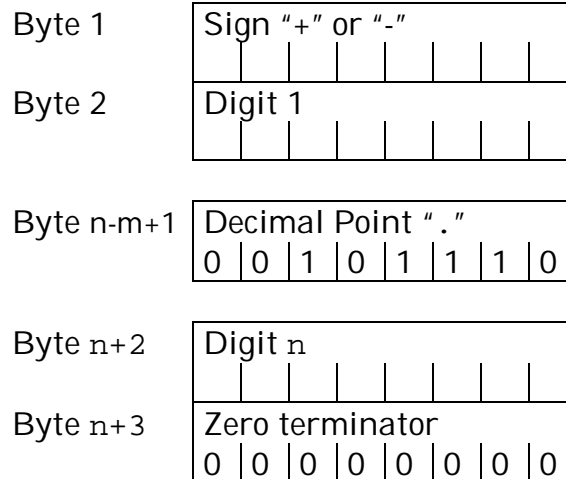
Size: 6 bytes

## Range

The range of ODBC DATE items is:

Maximum: December 31, 9999 AD

	Minimum:	January 1, 6000 BC
Examples	DEFINE ORD_DATE :	ODBC DATE
	Defines ORD_DATE as a 6 byte date in ODBC format.	
ODBC DECIMAL	The ODBC DECIMAL data type represents fixed point numeric data.	
Syntax	DECIMAL	or NUMERIC
	DECIMAL(n)	or NUMERIC(n)
	DECIMAL(n,m)	or NUMERIC(n,m)
	The keywords DECIMAL, DEC, and NUMERIC have identical meanings and may be used interchangeably.	
	n specifies the maximum number of digits the field may hold, including digits to the right of the decimal point. n must be from 1 to 38. If n is omitted, n is assumed to be 15.	
	If m is specified, m indicates the number of digits to the right of the decimal point the field may hold. If m is not specified, m is assumed to be 0. m must be from 0 to n.	
	ALLOW NULLS may be appended to the data type specification to allow storage of a null value.	
Display	The default print picture of ODBC DECIMAL fields is:	
	PIC "-(n-m)9.9(m)"	for m > 0
	PIC "-(n)9"	for n > 0
	PIC "-(9)9.9(6)"	for n = 0
Family	Numeric	
Technical	The byte layout of ODBC DECIMAL type items is as follows:	



ODBC DECIMAL type items require one byte per digit stored (n), plus one byte for the sign, plus one byte for the decimal point, plus one byte for a zero terminator.

Size: n + 3 bytes

## Range

The range of ODBC DECIMAL items is:

Maximum: +99..99 with n - m digits

Minimum: -99..99 with n - m digits

## Examples

```
DEFINE DEC : ODBC DECIMAL(6)
DEFINE AMT : ODBC NUMERIC(10,2)
```

Defines DEC as a 9 byte fixed decimal integer that can hold up to 6 digits. The default print picture for DEC is "-(6)9". Defines AMT as a 13 byte fixed decimal integer that can hold up to 10 digits: 8 digits to the left of the decimal point, and 2 digits to the right of the decimal point. The default print picture for AMT is "-(8)9.9(2)"

## ODBC DOUBLE PRECISION

The ODBC DOUBLE PRECISION data type represents an 8 byte IEEE floating point number.

## Syntax

ODBC DOUBLE PRECISION

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

Display	The default print picture of ODBC DOUBLE PRECISION fields is:  PIC "-(9)9.9(6)"
Family	Numeric, floating point
Technical	The byte layout of the ODBC DOUBLE PRECISION data type is as follows:

Byte 1	f <sub>44</sub>	f <sub>45</sub>	f <sub>46</sub>	f <sub>47</sub>	f <sub>48</sub>	f <sub>49</sub>	f <sub>50</sub>	f <sub>51</sub>
Byte 2	f <sub>36</sub>	f <sub>37</sub>	f <sub>38</sub>	f <sub>39</sub>	f <sub>40</sub>	f <sub>41</sub>	f <sub>42</sub>	f <sub>43</sub>
Byte 3	f <sub>28</sub>	f <sub>29</sub>	f <sub>30</sub>	f <sub>31</sub>	f <sub>32</sub>	f <sub>33</sub>	f <sub>34</sub>	f <sub>35</sub>
Byte 4	f <sub>20</sub>	f <sub>21</sub>	f <sub>22</sub>	f <sub>23</sub>	f <sub>24</sub>	f <sub>25</sub>	f <sub>26</sub>	f <sub>27</sub>
Byte 5	f <sub>12</sub>	f <sub>13</sub>	f <sub>14</sub>	f <sub>15</sub>	f <sub>16</sub>	f <sub>17</sub>	f <sub>18</sub>	f <sub>19</sub>
Byte 6	f <sub>4</sub>	f <sub>5</sub>	f <sub>6</sub>	f <sub>7</sub>	f <sub>8</sub>	f <sub>9</sub>	f <sub>10</sub>	f <sub>11</sub>
Byte 7	e <sub>7</sub>	e <sub>8</sub>	e <sub>9</sub>	e <sub>10</sub>	f <sub>0</sub>	f <sub>1</sub>	f <sub>2</sub>	f <sub>3</sub>
Byte 8	s	e <sub>0</sub>	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>

Size: 8 bytes

s = sign bit, 1 = negative

e = exponent, 11 bits, biased by 1023

f = fraction, 52 bits, implicit leading 1

Value:  $(-1)^s * 2^{(e - 1023)} * (1.f)$

Notes: Exponents of all 0's and all 1's are reserved. An exponent with all 0's and a fraction of all 0's represents the number 0. An exponent of all 0's and a non-zero fraction represents a denormalized number. An exponent of all 1's and a fraction of all 0's represents infinity. An exponent of all 1's and a non-zero fraction represents a NaN (not a number).

Range The range of ODBC DOUBLE PRECISION items is:

Maximum:  $+1.797693134862318 \cdot 10^{308}$

Minimum:  $-1.797693134862318 \cdot 10^{308}$

Minimum > 0:  $4.940656458412465 \cdot 10^{-324}$

Examples DEFINE FLT : ODBC DOUBLE PRECISION

Defines FLT as an IEEE 8 byte floating point number.

## ODBC INTEGER

The ODBC INTEGER data type represents a 4 byte binary signed integer in twos-complement form.

### Syntax

ODBC INTEGER or ODBC INT

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

### Display

The default print picture of ODBC INTEGER fields is:

PIC "-(10)9"

### Family

Numeric, binary integer

### Technical

The byte layout of the ODBC INTEGER data type is as follows:

Byte 1	b <sub>23</sub>	b <sub>24</sub>	b <sub>25</sub>	b <sub>26</sub>	b <sub>27</sub>	b <sub>28</sub>	b <sub>29</sub>	b <sub>30</sub>
Byte 2	b <sub>15</sub>	b <sub>16</sub>	b <sub>17</sub>	b <sub>18</sub>	b <sub>19</sub>	b <sub>20</sub>	b <sub>21</sub>	b <sub>22</sub>
Byte 3	b <sub>7</sub>	b <sub>8</sub>	b <sub>9</sub>	b <sub>10</sub>	b <sub>11</sub>	b <sub>12</sub>	b <sub>13</sub>	b <sub>14</sub>
Byte 4	s	b <sub>0</sub>	b <sub>1</sub>	b <sub>2</sub>	b <sub>3</sub>	b <sub>4</sub>	b <sub>5</sub>	b <sub>6</sub>

Size: 4 bytes

s = sign bit, 1 = negative

b<sub>n</sub> = b<sub>0</sub> is the most significant bit and b<sub>30</sub> is the least significant bit

### Range

The range of INTEGER items is:

Maximum: +2147483647

Minimum: -2147483648

### Examples

DEFINE INT : ODBC INTEGER

Defines INT as a 4 byte binary integer.

## ODBC LONG VARBINARY

The ODBC LONG VARBINARY data type represents variable length binary data.

Syntax ODBC LONG VARBINARY

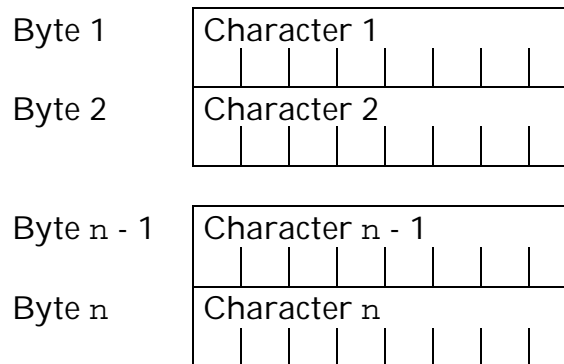
The absolute maximum number of bytes that can be represented with the ODBC LONG VARBINARY data type is 2,147,483,643. In practice, this limit can probably never be reached due to practical considerations.

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

Display ODBC LONG VARBINARY fields are printed in hexadecimal using a default print width of  $2 * (\text{length} + 1)$ .

Family Binary, variable length

Technical The byte layout of ODBC LONG VARBINARY items is as follows:



Size: n bytes, where n is the number of bytes required to store a particular item.

Examples `DEFINE BIN : ODBC LONG VARBINARY`

Defines BIN as a long variable length binary string.

ODBC LONG NVARCHAR The ODBC LONG NVARCHAR data type represents variable length native (double byte) character data.

Syntax ODBC LONG NVARCHAR

The absolute maximum number of bytes that can be represented with the ODBC LONG NVARCHAR data type is 1,073,741,823. In practice however, this limit can probably never be reached due to practical considerations.

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

Display

There is no default print width for ODBC LONG NVARCHAR fields. By default the entire field is printed.

Family

Character string, variable length, native

Technical

The byte layout of ODBC LONG NVARCHAR items is as follows:

Byte 1	Character 1 (L)
Byte 2	Character 1 (H)
Byte 3	Character 2 (L)
Byte 4	Character 2 (H)
Byte n*2-1	Character n (L)
Byte n*2	Character n (H)

Size:  $n * 2$  bytes, where  $n$  is the number of characters in the string.

Examples

DEFINE CH : ODBC LONG NVARCHAR

Defines CH as a long variable length native character string.

ODBC LONG VARCHAR

The ODBC LONG VARCHAR data type represents variable length character data.



## Syntax

ODBC LONG VARCHAR

The absolute maximum number of bytes that can be represented with the ODBC LONG VARCHAR data type is 2,147,483,643. In practice however, this limit can probably never be reached due to practical considerations.

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

## Display

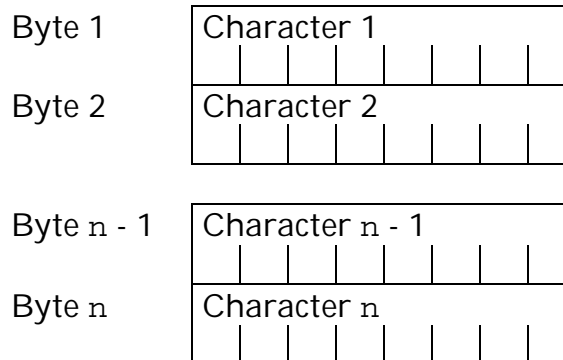
There is no default print width for ODBC LONG VARCHAR fields. By default the entire field is printed.

## Family

Character string, variable length

## Technical

The byte layout of ODBC LONG VARCHAR items is as follows:



Size: n bytes, where n is the number of bytes required to store a particular item.

## Examples

```
DEFINE CH : ODBC LONG VARCHAR
```

Defines CH as a long variable length character string.

## ODBC NCHAR

The ODBC NCHAR data type represents fixed length native (double byte) character data.

## Syntax

ODBC NCHAR(n)

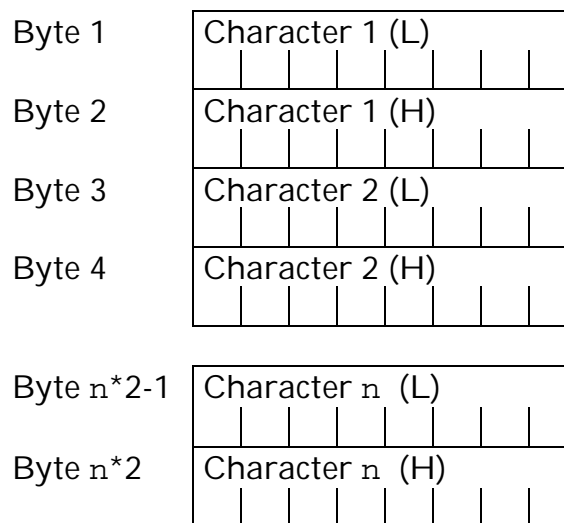
$n$  specifies the length in bytes of the field.  $n$  must be from 1 to 8000.

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

**Display** The default print width of ODBC NCHAR fields is the width of the field, or  $n$ .

**Family** Character string, fixed length, native

**Technical** The byte layout of ODBC NCHAR items is as follows:



Size:  $n * 2$  bytes

**Examples**   
`DEFINE CH : ODBC NCHAR(20) ALLOW NULLS`  
`DEFINE BIGCH : ODBC NCHAR(200)`

Defines CH as a 20 byte fixed length native character string that allows null values. Defines BIGCH as a 200 byte fixed length native character string.

**ODBC NVARCHAR** The ODBC NVARCHAR data type represents variable length native (double byte) character data.

**Syntax** `ODBC NVARCHAR( $n$ )`

$n$  specifies the maximum length in bytes of the

field.  $n$  must be from 1 to 8000.

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

Display

The default print width of ODBC NVARCHAR fields is the width of the field, or  $n$ .

Family

Character string, variable length, native

Technical

The byte layout of ODBC NVARCHAR items is as follows:

Byte 1	Character 1 (L)
Byte 2	Character 1 (H)
Byte 3	Character 2 (L)
Byte 4	Character 2 (H)
Byte $n*2-1$	Character $n$ (L)
Byte $n*2$	Character $n$ (H)
Byte $n*2+1$	Length byte (H)
Byte $n*2+2$	Length byte (L)

Size:  $(n * 2) + 2$  bytes

Examples

```
DEFINE CH : ODBC NVARCHAR(8) ALLOW NULLS
DEFINE BIGCH : ODBC NVARCHAR(2000)
```

Defines CH as a variable length native character string capable of holding up to 8 characters that allows null values. Defines BIGCH as a variable length native character string capable of holding up to 2000 characters.

ODBC REAL

The ODBC REAL data type represents a 4 byte IEEE floating point number.

Syntax

ODBC REAL

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

Display

The default print picture of ODBC REAL fields is:

PIC "-(9)9.9(6)"

Family

Numeric, floating point

Technical

The byte layout of the ODBC REAL data type is as follows:

Byte 1	f <sub>15</sub>	f <sub>16</sub>	f <sub>17</sub>	f <sub>18</sub>	f <sub>19</sub>	f <sub>20</sub>	f <sub>21</sub>	f <sub>22</sub>
Byte 2	f <sub>7</sub>	f <sub>8</sub>	f <sub>9</sub>	f <sub>10</sub>	f <sub>11</sub>	f <sub>12</sub>	f <sub>13</sub>	f <sub>14</sub>
Byte 3	e <sub>7</sub>	f <sub>0</sub>	f <sub>1</sub>	f <sub>2</sub>	f <sub>3</sub>	f <sub>4</sub>	f <sub>5</sub>	f <sub>6</sub>
Byte 4	s	e <sub>0</sub>	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>

Size: 4 bytes

s = sign bit, 1 = negative

e = exponent, 8 bits, biased by 127

f = fraction, 23 bits, implicit leading 1

Value:  $(-1)^s * 2^{(e - 127)} * (1.f)$

Notes: Exponents of all 0's and all 1's are reserved. An exponent with all 0's and a fraction of all 0's represents the number 0. An exponent of all 0's and a non-zero fraction represents a denormalized number. An exponent of all 1's and a fraction of all 0's represents infinity. An exponent of all 1's and a non-zero fraction represents a NaN (not a number).

Range

The range of ODBC REAL items is:

Maximum:  $+3.402823 \cdot 10^{38}$

Minimum:  $-3.402823 \cdot 10^{38}$

Minimum > 0:  $+1.175495 \cdot 10^{-38}$

Examples

DEFINE FLT : ODBC REAL

Defines `FLT` as an IEEE 4 byte floating point number.

ODBC `SMALLINT`

The ODBC `SMALLINT` data type represents a 2 byte binary signed integer in twos-complement form.

## Syntax

ODBC `SMALLINT`

`ALLOW NULLS` may be appended to the data type specification to allow storage of a null value.

## Display

The default print picture of ODBC `SMALLINT` fields is:

`PIC "-(5)9"`

## Family

Numeric, binary integer

## Technical

The byte layout of the ODBC `SMALLINT` data type is as follows:

Byte 1	b <sub>7</sub>	b <sub>8</sub>	b <sub>9</sub>	b <sub>10</sub>	b <sub>11</sub>	b <sub>12</sub>	b <sub>13</sub>	b <sub>14</sub>
Byte 2	s	b <sub>0</sub>	b <sub>1</sub>	b <sub>2</sub>	b <sub>3</sub>	b <sub>4</sub>	b <sub>5</sub>	b <sub>6</sub>

Size: 2 bytes

s = sign bit, 1 = negative

b<sub>n</sub> = b<sub>0</sub> is the most significant bit and b<sub>14</sub> is the least significant bit

## Range

The range of ODBC `SMALLINT` items is:

Maximum: +32767

Minimum: -32768

## Examples

`DEFINE INT : ODBC SMALLINT`

Defines `INT` as a 2 byte binary integer.

ODBC `TIME`

The ODBC `TIME` data type represents a date and/or time.

## Syntax

ODBC `TIME`

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

**Display** The default print width of ODBC TIME fields is 8 in the following format:  
hh:mm:ss

**Family** Date/time

**Technical** The byte layout of ODBC TIME items is as follows:

Byte 1	Hour
Byte 2	Hour filler (zero)
Byte 3	Minute
Byte 4	Minute filler (zero)
Byte 5	Second
Byte 6	Second filler (zero)

Size: 6 bytes

**Range** The range of ODBC TIME items is:

Maximum: 23:59:59  
Minimum: 00:00:00

**Examples** DEFINE START\_TIME : ODBC TIME  
  
Defines START\_TIME as a 16 byte time in ODBC format.

**ODBC TIMESTAMP** The ODBC TIMESTAMP data type represents a date and/or time.

**Syntax** ODBC TIMESTAMP

ALLOW NULLS may be appended to the data type

specification to allow storage of a null value.

#### Display

The default print width of ODBC `TIMESTAMP` fields is 20 in the following format:

`dd-mmm-yyyy hh:mm:ss`

#### Family

Date/time

#### Technical

The byte layout of ODBC `TIMESTAMP` items is as follows:

Byte 1	Year low bits 
Byte 2	Year high bits 
Byte 3	Month 
Byte 4	Month filler (zero) 
Byte 5	Day 
Byte 6	Day filler (zero) 
Byte 7	Hour 
Byte 8	Hour filler (zero) 
Byte 9	Minute 
Byte 10	Minute filler (zero) 
Byte 11	Second 
Byte 12	Second filler (zero) 
Byte 13	Nanosecond 4 (Low) 
Byte 14	Nanosecond 3 
Byte 15	Nanosecond 2 
Byte 16	Nanosecond 1 (High) 

	Size: 16 bytes								
Range	<p>The range of ODBC <code>TIMESTAMP</code> items is:</p> <p>Maximum: December 31, 9999 AD 23:59 Minimum: January 1, 6000 BC 00:00</p>								
Examples	<p><code>DEFINE ORD_DATE : ODBC TIMESTAMP</code></p> <p>Defines <code>ORD_DATE</code> as a 16 byte date in ODBC format.</p>								
ODBC <code>TINYINT</code>	The ODBC <code>TINYINT</code> data type represents a 1 byte binary signed integer in twos-complement form.								
Syntax	<p><code>ODBC TINYINT</code></p> <p><code>ALLOW NULLS</code> may be appended to the data type specification to allow storage of a null value.</p>								
Display	<p>The default print picture of ODBC <code>SMALLINT</code> fields is:</p> <p><code>PIC "-(3)9"</code></p>								
Family	Numeric, binary integer								
Technical	<p>The byte layout of the ODBC <code>TINYINT</code> data type is as follows:</p> <p>Byte 1</p> <table><tr><td>s</td><td>b<sub>0</sub></td><td>b<sub>1</sub></td><td>b<sub>2</sub></td><td>b<sub>3</sub></td><td>b<sub>4</sub></td><td>b<sub>5</sub></td><td>b<sub>6</sub></td></tr></table> <p>Size: 1 byte</p> <p>s = sign bit, 1 = negative b<sub>n</sub> = b<sub>0</sub> is the most significant bit and b<sub>6</sub> is the least significant bit</p>	s	b <sub>0</sub>	b <sub>1</sub>	b <sub>2</sub>	b <sub>3</sub>	b <sub>4</sub>	b <sub>5</sub>	b <sub>6</sub>
s	b <sub>0</sub>	b <sub>1</sub>	b <sub>2</sub>	b <sub>3</sub>	b <sub>4</sub>	b <sub>5</sub>	b <sub>6</sub>		
Range	<p>The range of ODBC <code>TINYINT</code> items is:</p> <p>Maximum: +127 Minimum: -128</p>								
Examples	<code>DEFINE INT : ODBC TINYINT</code>								



Defines INT as a 1 byte binary integer.

#### ODBC UNIQUEIDENTIFIER

The ODBC UNIQUEIDENTIFIER data type represents 16 bytes of binary data. Its purpose is to provide support for the SQL Server **uniqueidentifier** data type which is used to contain a globally unique value.

#### Syntax

ODBC UNIQUEIDENTIFIER

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

#### Display

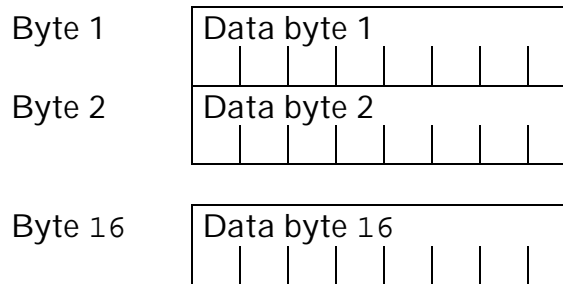
ODBC UNIQUEIDENTIFIER fields are printed in hexadecimal using a default print width of 34.

#### Family

Binary, fixed length

#### Technical

The byte layout of ODBC BINARY items is as follows:



Size: 16 bytes

#### Example

```
DEFINE UNQID : ODBC UNIQUEIDENTIFIER
```

Defines UNQID as a 16 byte binary field.

#### ODBC VARBINARY

The ODBC VARBINARY data type represents variable length binary data.

#### Syntax

ODBC VARBINARY(*n*)

*n* specifies the maximum length in bytes of the field. *n* must be from 1 to 8000.

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

Display

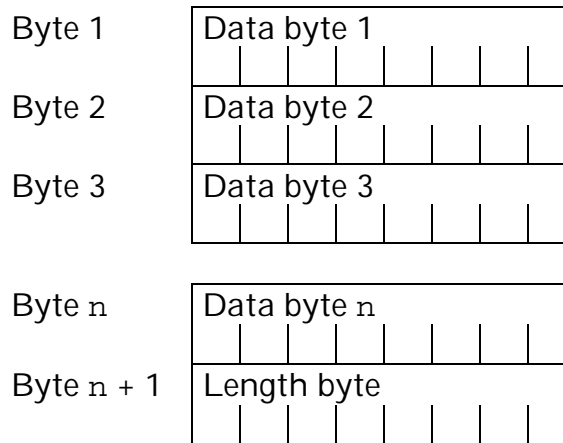
ODBC VARBINARY printed in hexadecimal using a default print width of  $2 * (n + 1)$ .

Family

Binary, variable length

Technical

The byte layout of ODBC VARBINARY items is as follows:



Size:  $n + 1$  bytes

Examples

```
DEFINE BIN : ODBC VARBINARY(20)
DEFINE BIGBIN : ODBC VARBINARY(200)
```

Defines BIN as a variable length binary string capable of holding up to 20 bytes. Defines BIGBIN as a variable length binary string capable of holding up to 200 bytes.

ODBC VARCHAR

The ODBC VARCHAR data type represents variable length character data.

Syntax

ODBC VARCHAR( $n$ )

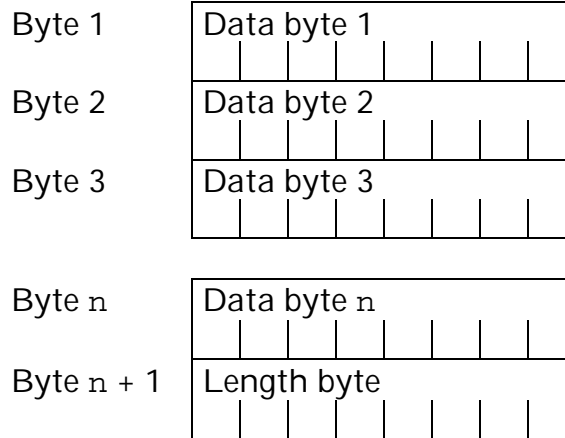
$n$  specifies the maximum length in bytes of the field.  $n$  must be from 1 to 8000.

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

**Display** The default print width of ODBC VARCHAR fields is the width of the field, or n.

**Family** Character string, variable length

**Technical** The byte layout of ODBC VARCHAR items is as follows:



Size: n + 1 bytes

**Examples** `DEFINE CH : ODBC VARCHAR(20) ALLOW NULLS`  
`DEFINE BIGCH : ODBC VARCHAR(2000)`

Defines CH as a variable length character string capable of holding up to 20 characters that allows null values. Defines BIGCH as a variable length character string capable of holding up to 2000 characters.

**Oracle Data Types**

The following data types originated on the database management system Oracle. The Oracle data types are all based on the internal Oracle representation.

Supported Oracle data types are as follows:

CHAR	Fixed length character data
DATE	Dates and time data
FLOAT	Interprets data as a ORACLE NUMBER ( 126 , -127 )
INTERVAL	Difference between two dates
LONG	Long variable length character data
LONG RAW	Long variable length binary data
NUMBER	Fixed point numeric data
RAW	Fixed length binary data
VARCHAR2	Variable length character data

**Null Values**

Oracle data types support null values. To indicate a data type that allows null values, `ALLOW NULLS` is appended to the data type specification.

Example:

```
DEFINE CV : ORACLE CHAR(20) ALLOW NULLS
```

The following is a detailed description of each of the Oracle data types supported by Warehouse.

**ORACLE CHAR**

The ORACLE CHAR data type represents fixed length character data.

**Syntax**

```
ORACLE CHAR (n)
```

`n` specifies the length in bytes of the field. `n` must be from 1 to 2000.

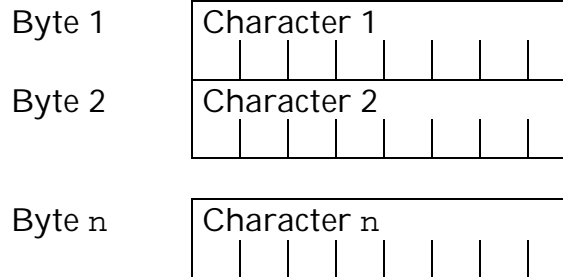
`ALLOW NULLS` may be appended to the data type specification to allow storage of a null value.

**Display**

The default print width of ORACLE CHAR fields is the width of the field, or `n`.

Family Character string, fixed length

Technical The byte layout of ORACLE CHAR items is as follows:



Size: n bytes

Examples

```
DEFINE CH : ORACLE CHAR(20) ALLOW NULLS
DEFINE BIGCH : ORACLE CHAR(200)
```

Defines CH as a 20 byte fixed length character string that allows null values. Defines BIGCH as a 200 byte fixed length character string.

ORACLE DATE The ORACLE DATE data type represents a date and/or time.

Syntax ORACLE DATE

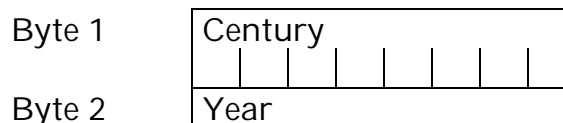
ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

Display The default print width of ORACLE DATE fields is 20 in the following format:

dd-mmm-yyyy hh:mm:ss

Family Date/time

Technical The byte layout of ORACLE DATE items is as follows:



Byte 3	Month								
Byte 4	Day								
Byte 5	Hour								
Byte 6	Minute								
Byte 7	Second								

The century byte and the year byte are biased by 100. All other bytes are biased by 1. The default time is midnight (0:00:00). For example, the date and time of May 8, 1995, 2:56:08 PM is stored as follows:

Byte 1:	119	19 + 100
Byte 2:	195	95 + 100
Byte 3:	6	5 + 1
Byte 4:	9	8 + 1
Byte 5:	15	14 + 1
Byte 6:	57	56 + 1
Byte 7:	9	8 + 1

Size: 7 bytes

#### Range

The range of ORACLE DATE items is:

Maximum:	December 31, 9999 AD
Minimum:	January 1, 6000 BC

#### Examples

```
DEFINE ORD_DATE : ORACLE DATE
```

Defines ORD\_DATE as a 7 byte date in Oracle format.

#### ORACLE FLOAT

The ORACLE FLOAT data type represents floating point decimal data.

#### Syntax

```
ORACLE FLOAT
```

	Interpreted as <code>ORACLE NUMBER(126,-127)</code> .
Display	See <code>ORACLE NUMBER</code> .
Family	Numeric
Technical	See <code>ORACLE NUMBER</code> .
Examples	<pre>DEFINE ORA_FLT : ORACLE FLOAT VALUE 3.1415926 PRINT ORA_FLT GO</pre> <p>Displays the following results:</p> <pre>3.141593</pre>
<code>ORACLE INTERVAL</code>	The <code>ORACLE INTERVAL</code> data type represents a numeric date interval.
Syntax	<pre>ORACLE INTERVAL DAY TO SECOND</pre> <p>or</p> <pre>ORACLE INTERVAL YEAR TO MONTH</pre> <p><code>DAY TO SECOND</code> is an interval containing days down to nanoseconds. Date arithmetic operations are available for this data type.</p> <p><code>YEAR TO MONTH</code> is an interval containing only the year and month. No date arithmetic operations are available for this data type.</p>
Display	<p>The interval can be printed using <code>PIC</code> elements to access just the numeric portion, just the time portion, or both.</p> <pre>PIC "N*" PIC "HH24:MI" PIC "N* HH24:MI:SS" PIC "N* HH24:MI:SS.T*"</pre> <p><code>N*</code> – Interval is interpreted as a simple number</p>

The numeric portion must always be accessed using the N\*, otherwise an Invalid date format specification error is thrown. The time portion can have any valid time format element.

Family

Date/time

Technical

The byte layout of ORACLE INTERVAL DAY TO SECOND items are as follows:

Byte 1	Digit 1
Byte 2	Digit 2
Byte 3	Digit 3
Byte 4	Digit 4
Byte 5	Hour
Byte 6	Minute
Byte 7	Second
Byte 8	Nanosecond 1
Byte 9	Nanosecond 2
Byte 10	Nanosecond 3
Byte 11	Nanosecond 4

Size: 11 bytes

The byte layout of ORACLE INTERVAL YEAR TO MONTH items are as follows:

Byte 1	Year 1
Byte 2	Year 2



Byte 3	Year 3
Byte 4	Year 4
Byte 5	Month

Size: 5 bytes

### Examples

```
DEFINE ORD_A      : ORACLE DATE
DEFINE ORD_B      : ORACLE DATE
DEFINE ORD_INT    : ORACLE INTERVAL DAY TO
                  SECOND
```

```
SETVAR ORD_A      = 20080101
SETVAR ORD_B      = 20081031
SETVAR ORD_INT    = ORD_B - ORD_A
```

```
PRINT ORD_A, ORD_B, ORD_INT
PRINT ORD_INT PIC "HH:MM:SS"
PRINT ORD_INT PIC "N*"
GO
```

Displays the following results:

```
01-JAN-2008  31-OCT-2008  304 00:00:00
12:00:00
304
```

### ORACLE LONG

The ORACLE LONG data type represents variable length character data.

### Syntax

```
ORACLE LONG
```

The absolute maximum number of bytes that can be represented with the ORACLE LONG data type is 2,147,483,643. In practice however, this limit can probably never be reached due to practical considerations. On some machines Warehouse uses the type ORACLE LONG\_ instead of ORACLE LONG. The difference between the two is the byte ordering of the length bytes. With ORACLE LONG, the high order byte is first. With ORACLE LONG\_, the high order byte is last.

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

Display

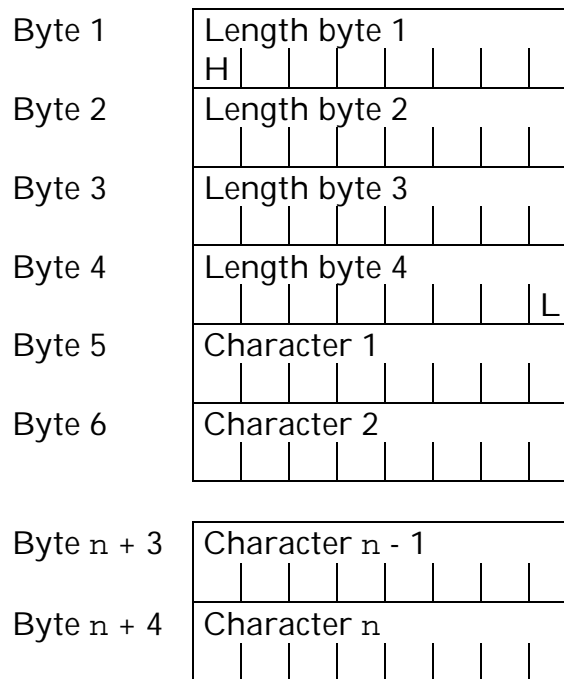
There is no default print width for ORACLE LONG fields. By default the entire field is printed.

Family

Character string, variable length

Technical

The byte layout of ORACLE LONG items is as follows:

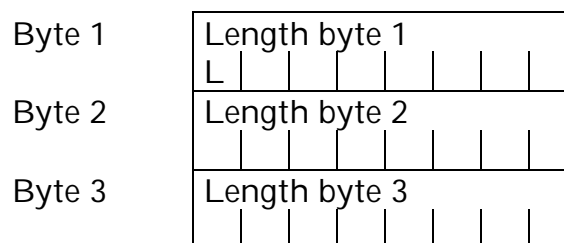


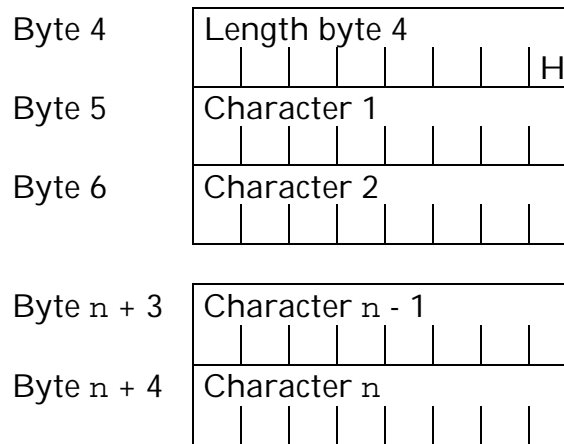
Size:  $n + 4$  bytes, where  $n$  is the number of bytes required to store a particular item.

H= High order bit

L= Low order bit

The byte layout of ORACLE LONG\_ items is as follows:





Size:  $n + 4$  bytes, where  $n$  is the number of bytes required to store a particular item.

H= High order bit

L= Low order bit

### Examples

```
DEFINE CH : ORACLE LONG ALLOW NULLS
```

Defines CH as a long variable length character string that allows null values.

### ORACLE LONG RAW

The ORACLE LONG RAW data type represents variable length binary data.

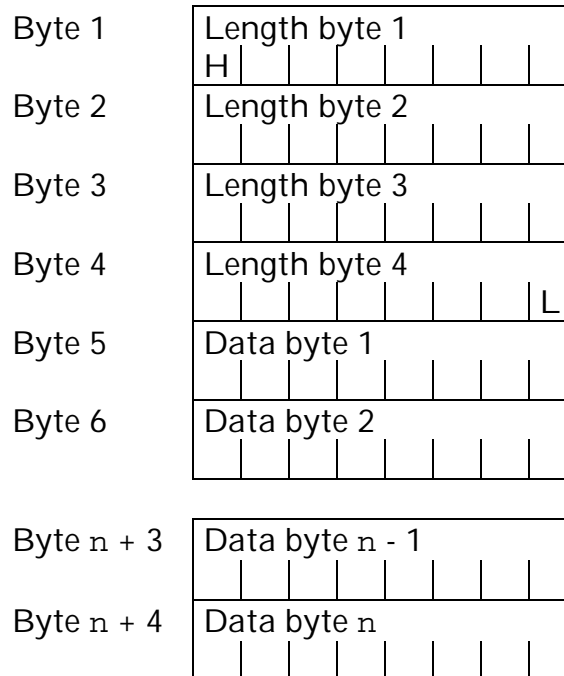
### Syntax

```
ORACLE LONG RAW
```

The absolute maximum number of bytes that can be represented with the ORACLE LONG RAW data type is 2,147,483,643. In practice, this limit can probably never be reached due to practical considerations. On some machines Warehouse uses the type ORACLE LONG RAW\_ instead of ORACLE LONG RAW. The difference between the two is the byte ordering of the length bytes. With ORACLE LONG RAW, the high order byte is first. With ORACLE LONG RAW\_, the high order byte is last.

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

Display	ORACLE LONG RAW fields are printed in hexadecimal using a default print width of $2 * (\text{length} + 1)$ .
Family	Binary, variable length
Technical	The byte layout of ORACLE LONG RAW items is as follows:

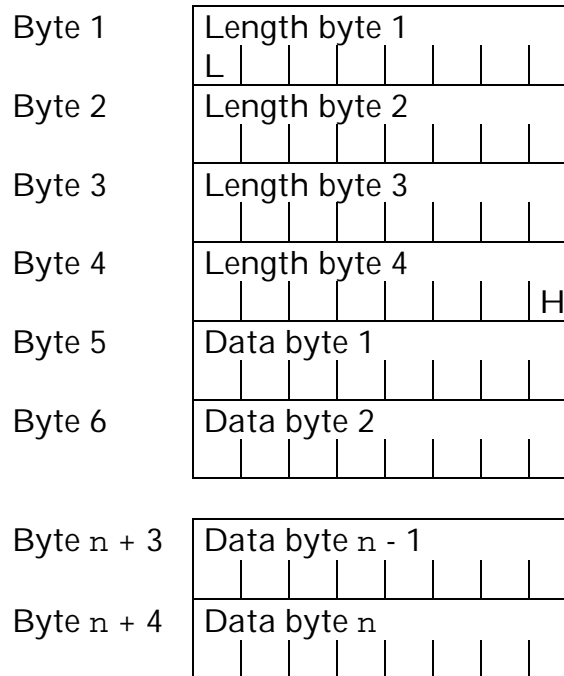


Size:  $n + 4$  bytes, where  $n$  is the number of bytes required to store a particular item.

H= High order bit

L= Low order bit

The byte layout of ORACLE LONG RAW\_ items is as follows:



Size:  $n + 4$  bytes, where  $n$  is the number of bytes required to store a particular item.

H= High order bit

L= Low order bit

### Examples

```
DEFINE BIN : ORACLE LONG RAW
```

Defines BIN as a long variable length binary string.

### ORACLE NUMBER

The ORACLE NUMBER data type represents floating point decimal data.

### Syntax

```
ORACLE NUMBER
```

or

```
ORACLE NUMBER (n)
```

or

```
ORACLE NUMBER (n, m)
```

$n$  specifies the maximum number of digits the field may hold, including digits to the right of the decimal point.  $n$  must be from 0 to 38. The default is 0 and indicates the maximum number of digits (38).

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

If  $m$  is specified,  $m$  indicates the maximum exponent of the number.  $m$  must be in the range of -84 to 127.

#### Display

The default print picture of ORACLE NUMBER fields is:

```
PIC "-(n-m)9.9(m)"    for m > 0
PIC "-(n)9"            for n > 0
PIC "-(9)9.9(6)"       for n = 0
```

#### Family

Numeric

#### Technical

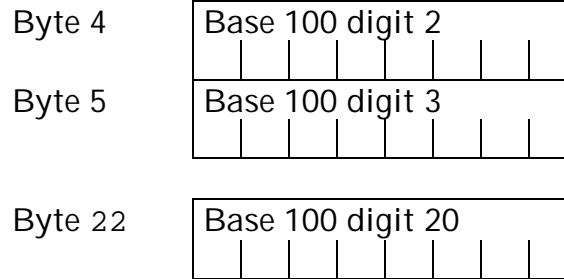
ORACLE NUMBER fields require 22 bytes of storage. The first byte contains a number that represents the number of bytes required to store the number. The length is from 1 to 21.

The second byte contains the sign and exponent of the number. The first bit contains the sign, 1 if positive. The remaining 7 bits contain the exponent. If the number is positive, 64 is added to the exponent. If the number is negative, the exponent is subtracted from 63.

The remaining bytes contain the digits of the number in base 100. For positive numbers, 1 is added to each base 100 digit. For negative numbers, the base 100 digit is subtracted from 101. Negative numbers less than 22 bytes long are terminated with 102. Each base 100 digit contains 2 base 10 digits.

Byte layout for an ORACLE NUMBER:

Byte 1	Length of number
Byte 2	Sign and Exponent
Byte 3	Base 100 digit 1



Size: 22 bytes

### Range

The range of ORACLE NUMBERS items is:

Maximum:  $+9.999999 \cdot 10^{125}$

Minimum:  $+9.999999 \cdot 10^{125}$

Minimum > 0:  $+1.000000 \cdot 10^{-129}$

### Examples

```
DEFINE DEC : ORACLE NUMBER(6)
DEFINE AMT : ORACLE NUMBER(10,2)
```

Defines DEC as an integer that can hold up to 6 digits. Defines AMT as a number that can hold up to 10 digits: 8 digits to the left of the decimal point, and 2 digits to the right of the decimal point.

### ORACLE RAW

The ORACLE RAW data type represents fixed length binary data.

### Syntax

ORACLE RAW(*n*)

*n* specifies the maximum length in bytes of the field. *n* must be from 1 to 2000.

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

### Display

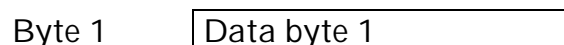
ORACLE RAW fields are printed in hexadecimal using a default print width of  $2 * (n + 1)$ .

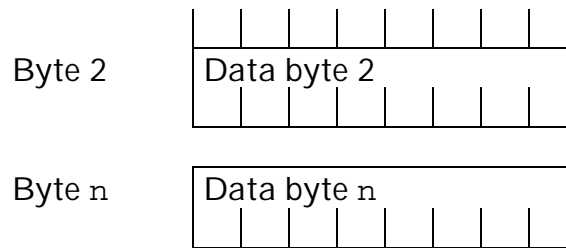
### Family

Binary, fixed length

### Technical

The byte layout of ORACLE RAW items is as follows:





Size: n bytes

#### Examples

```
DEFINE BIN : ORACLE RAW(20)
DEFINE BIGBIN : ORACLE RAW(200)
```

Defines BIN as a 20 byte binary field. Defines BIGBIN as a 200 byte binary field.

#### ORACLE TIMESTAMP

The ORACLE TIMESTAMP data type represents a date and/or time with precision to the nanosecond.

#### Syntax

```
ORACLE TIMESTAMP [ALLOW NULLS]
```

or

```
ORACLE TIMESTAMP WITH [LOCAL] TIME ZONE
[ALLOW NULLS]
```

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

WITH [LOCAL] TIME ZONE when specified will store time zone information in addition to the date and time. If local is specified, assumes the local time zone.

#### Display

The default print width of ORACLE TIMESTAMP fields is 20 in the following format:

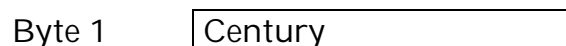
```
dd-mmm-yyyy hh:mm:ss
```

#### Family

Date/time

#### Technical

The byte layout of ORACLE TIMESTAMP items is as follows:



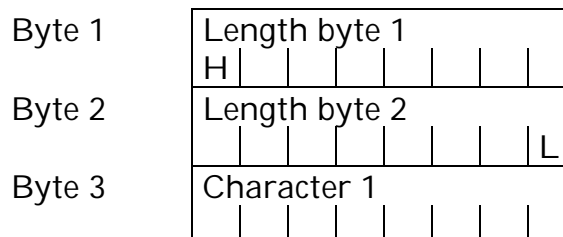


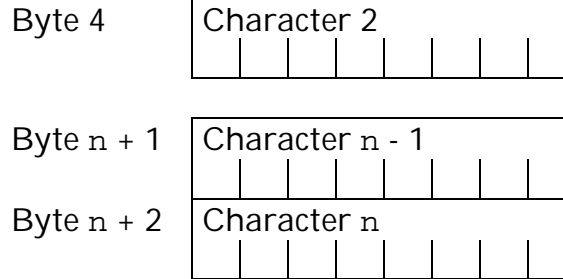
Byte 2	Year										
Byte 3	Month										
Byte 4	Day										
Byte 5	Hour										
Byte 6	Minute										
Byte 7	Second										
Byte 8	Nanosecond 1										
Byte 9	Nanosecond 2										
Byte 10	Nanosecond 3										
Byte 11	Nanosecond 4										

The century byte and the year byte are biased by 100. All other bytes are biased by 1. The nanoseconds are stored in bytes 8-11 as a 32 bit binary integer with byte eight being the high order byte and byte 11 being the low order byte. The default time is midnight (0:00:00). For example, the date and time of May 8, 1995, 2:56:08.128 PM is stored as follows:

Byte 1:	119	19 + 100
Byte 2:	195	95 + 100
Byte 3:	6	5 + 1
Byte 4:	9	8 + 1
Byte 5:	15	14 + 1
Byte 6:	57	56 + 1
Byte 7:	9	8 + 1
Byte 8:	7	1 <sup>st</sup> byte of 128,000,000
Byte 9:	161	2 <sup>nd</sup> byte of 128,000,000
Byte 10:	32	3 <sup>rd</sup> byte of 128,000,000
Byte 11:	0	4 <sup>th</sup> byte of 128,000,000

	Size: 11 bytes
Range	<p>The range of ORACLE DATE items is:</p> <p>Maximum: December 31, 9999 AD</p> <p>Minimum: January 1, 6000 BC</p>
Examples	<p>DEFINE ORD_TS : ORACLE TIMESTAMP</p> <p>Defines ORD_TS as an 11 byte timestamp in Oracle format.</p>
ORACLE VARCHAR2	The ORACLE VARCHAR2 data type represents variable length character data.
Syntax	<p>ORACLE VARCHAR2(n)</p> <p>n specifies the maximum length in bytes of the field. n must be from 1 to 4000. On some machines Warehouse uses the type ORACLE VARCHAR2_ instead of ORACLE VARCHAR2. The difference between the two is the byte ordering of the length bytes. With ORACLE VARCHAR2, the high order byte is first. With ORACLE VARCHAR2_, the high order byte is last.</p> <p>ALLOW NULLS may be appended to the data type specification to allow storage of a null value.</p>
Display	The default print width of ORACLE VARCHAR2 fields is the width of the field, or n.
Family	Character string, variable length
Technical	The byte layout of ORACLE VARCHAR2 items is as follows:



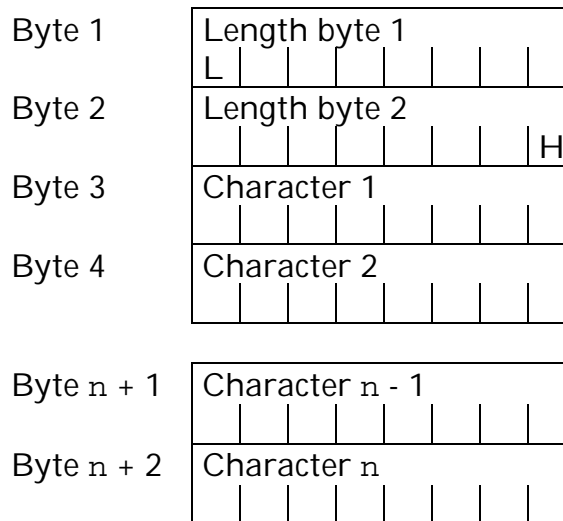


Size: n + 2 bytes

H= High order bit

L= Low order bit

The byte layout of ORACLE VARCHAR2\_ items is as follows:



Size: n + 2 bytes

H= High order bit

L= Low order bit

### Examples

```
DEFINE CH : ORACLE VARCHAR2(20)
DEFINE BIGCH : ORACLE VARCHAR2(1000)
```

Defines CH as a variable length character string capable of holding up to 20 characters. Defines BIGCH as a variable length character string capable of holding up to 1000 characters.

**SQL Data Types**

The following data types originated from the DB2 database management system from IBM.

Supported SQL data types are as follows:

BINARY	Fixed length binary data
CHAR	Fixed length character data
DATE	Calendar date
DECIMAL	Fixed point numeric data
DOUBLE PRECISION	64 bit IEEE floating point data
INTEGER	32 bit integer data
LONG VARBINARY	Long variable length binary data
LONG VARCHAR	Long variable length character data
NUMERIC	Alias for DECIMAL
REAL	32 bit IEEE floating point data
SMALLINT	16 bit integer data
TIME	Time of day
TIMESTAMP	Calendar date and time
VARBINARY	Variable length binary data
VARCHAR	Variable length character data

**Null Values**

SQL data types support null values. To indicate a data type that allows null values, `ALLOW NULLS` is appended to the data type specification. Example:

```
DEFINE CV : SQL CHAR(20) ALLOW NULLS
```

The following is a detailed description of each of the SQL data types supported by Warehouse.

**SQL BINARY**

The `SQL BINARY` data type represents fixed length binary data.

**Syntax**

```
SQL BINARY(n)
```

`n` specifies the length in bytes of the field. `n` must

be from 1 to 254.

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

Display

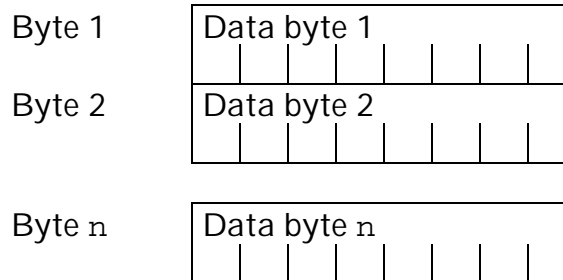
SQL BINARY fields are printed in hexadecimal using a default print width of  $2 * (n + 1)$ .

Family

Binary, fixed length

Technical

The byte layout of SQL BINARY items is as follows:



Size: n bytes

Examples

```
DEFINE BIN : SQL BINARY(20)
DEFINE BIGBIN : SQL BINARY(2000)
```

Defines BIN as a 20 byte binary field. Defines BIGBIN as a 2000 byte binary field.

SQL CHAR

The SQL CHAR data type represents fixed length character data.

Syntax

SQL CHAR(n)

n specifies the length in bytes of the field. n must be from 1 to 254.

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

Display

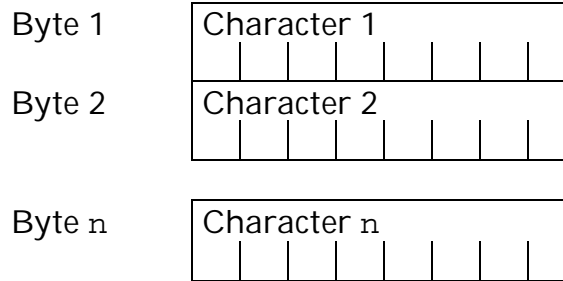
The default print width of SQL CHAR fields is the width of the field, or n.

Family

Character string, fixed length

## Technical

The byte layout of SQL CHAR items is as follows:



Size: n bytes

## Examples

```
DEFINE CH : SQL CHAR(20)
DEFINE BIGCH : SQL CHAR(200)
```

Defines CH as a 20 byte fixed length character string. Defines BIGCH as a 200 byte fixed length character string.

## SQL DATE

The SQL DATE data type represents a date and/or time.

## Syntax

SQL DATE

On some machines Warehouse uses the type SQL DATE\_ instead of SQL DATE. The difference between the two is the data alignment. SQL DATE fields may be 1 byte aligned. SQL DATE\_ fields are 2 byte aligned.

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

## Display

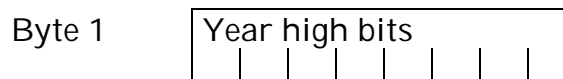
The default print width of SQL DATE fields is 11 in the following format:  
dd-mmm-yyyy

## Family

Date/time

## Technical

The byte layout of SQL DATE items is as follows:



Byte 2	Year low bits
Byte 3	Month filler (zero)
Byte 4	Month
Byte 5	Day filler (zero)
Byte 6	Day

Size: 6 bytes

### Range

The range of SQL DATE items is:

Maximum: December 31, 9999 AD

Minimum: January 1, 6000 BC

### Examples

```
DEFINE ORD_DATE : SQL DATE ALLOW NULLS
```

Defines ORD\_DATE as a 6 byte date in SQL format that allows null values.

## SQL DECIMAL

The SQL DECIMAL data type represents fixed point numeric data.

### Syntax

```
DECIMAL          or      NUMERIC
DECIMAL(n)       or      NUMERIC(n)
DECIMAL(n,m)     or      NUMERIC(n,m)
```

The keywords DECIMAL, DEC, and NUMERIC have identical meanings and may be used interchangeably.

*n* specifies the maximum number of digits the field may hold, including digits to the right of the decimal point. *n* must be from 1 to 28. If *n* is omitted, *n* is assumed to be 15.

If *m* is specified, *m* indicates the number of digits to the right of the decimal point the field may hold. If *m* is not specified, *m* is assumed to be 0. *m* must be from 0 to *n*.

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

## Display

The default print picture of SQL DECIMAL fields is:

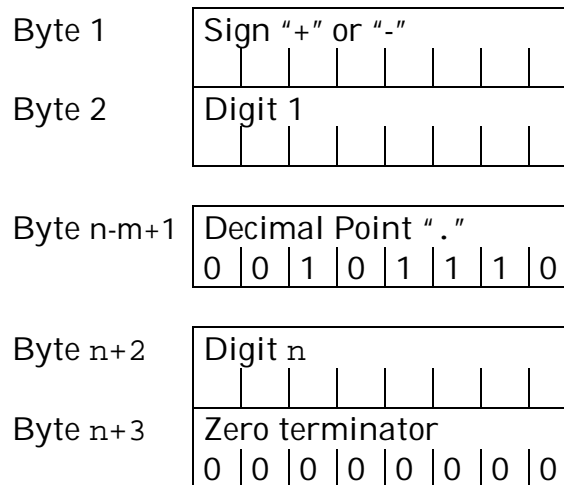
```
PIC "-(n-m)9.9(m)"    for m > 0
PIC "-(n)9"            for n > 0
PIC "-(9)9.9(6)"      for n = 0
```

## Family

Numeric

## Technical

The byte layout of SQL DECIMAL type items is as follows:



SQL DECIMAL type items require one byte per digit stored (n), plus one byte for the sign, plus one byte for the decimal point, plus one byte for a zero terminator.

Size:  $n + 3$  bytes

## Range

The range of SQL DECIMAL items is:

```
Maximum:    +99..99 with n - m digits
Minimum:    -99..99 with n - m digits
```

## Examples

```
DEFINE DEC : SQL DECIMAL(6)
DEFINE AMT : SQL NUMERIC(10,2)
```

Defines DEC as a 9 byte fixed decimal integer that can hold up to 6 digits. The default print picture



for DEC is "-(6)9". Defines AMT as a 13 byte fixed decimal integer that can hold up to 10 digits: 8 digits to the left of the decimal point, and 2 digits to the right of the decimal point. The default print picture for AMT is "-(8)9.9(2)"

SQL DOUBLE  
PRECISION

The SQL DOUBLE PRECISION data type represents an 8 byte IEEE floating point number.

Syntax

SQL DOUBLE PRECISION

On some machines Warehouse uses the type SQL DOUBLE PRECISION\_ instead of SQL DOUBLE PRECISION. The difference between the two is the data alignment. SQL DOUBLE PRECISION fields may be 1 byte aligned. SQL DOUBLE PRECISION\_ fields are 8 byte aligned.

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

Display

The default print picture of SQL DOUBLE PRECISION fields is:

PIC "-(9)9.9(6)"

Family

Numeric, floating point

Technical

The byte layout of the SQL DOUBLE PRECISION data type is as follows:

Byte 1	s	e <sub>0</sub>	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	e <sub>6</sub>
Byte 2	e <sub>7</sub>	e <sub>8</sub>	e <sub>9</sub>	e <sub>10</sub>	f <sub>0</sub>	f <sub>1</sub>	f <sub>2</sub>	f <sub>3</sub>
Byte 3	f <sub>4</sub>	f <sub>5</sub>	f <sub>6</sub>	f <sub>7</sub>	f <sub>8</sub>	f <sub>9</sub>	f <sub>10</sub>	f <sub>11</sub>
Byte 4	f <sub>12</sub>	f <sub>13</sub>	f <sub>14</sub>	f <sub>15</sub>	f <sub>16</sub>	f <sub>17</sub>	f <sub>18</sub>	f <sub>19</sub>
Byte 5	f <sub>20</sub>	f <sub>21</sub>	f <sub>22</sub>	f <sub>23</sub>	f <sub>24</sub>	f <sub>25</sub>	f <sub>26</sub>	f <sub>27</sub>
Byte 6	f <sub>28</sub>	f <sub>29</sub>	f <sub>30</sub>	f <sub>31</sub>	f <sub>32</sub>	f <sub>33</sub>	f <sub>34</sub>	f <sub>35</sub>
Byte 7	f <sub>36</sub>	f <sub>37</sub>	f <sub>38</sub>	f <sub>39</sub>	f <sub>40</sub>	f <sub>41</sub>	f <sub>42</sub>	f <sub>43</sub>
Byte 8	f <sub>44</sub>	f <sub>45</sub>	f <sub>46</sub>	f <sub>47</sub>	f <sub>48</sub>	f <sub>49</sub>	f <sub>50</sub>	f <sub>51</sub>

Size: 8 bytes

s = sign bit, 1 = negative

e = exponent, 11 bits, biased by 1023  
 f = fraction, 52 bits, implicit leading 1

Value:  $(-1)^s * 2^{(e - 1023)} * (1.f)$

Notes: Exponents of all 0's and all 1's are reserved.  
 An exponent with all 0's and a fraction of all 0's represents the number 0. An exponent of all 0's and a non-zero fraction represents a denormalized number. An exponent of all 1's and a fraction of all 0's represents infinity. An exponent of all 1's and a non-zero fraction represents a NaN (not a number).

#### Range

The range of SQL DOUBLE PRECISION items is:

Maximum:  $+1.797693134862318 \cdot 10^{308}$   
 Minimum:  $-1.797693134862318 \cdot 10^{308}$   
 Minimum > 0:  $4.940656458412465 \cdot 10^{-324}$

#### Examples

```
DEFINE FLT : SQL DOUBLE PRECISION
```

Defines FLT as an IEEE 8 byte floating point number.

#### SQL INTEGER

The SQL INTEGER data type represents a 4 byte binary signed integer in twos-complement form.

#### Syntax

SQL INTEGER or SQL INT

On some machines Warehouse uses the type SQL INTEGER\_ instead of SQL INTEGER. The difference between the two is the data alignment. SQL INTEGER fields may be 1 byte aligned. SQL INTEGER\_ fields are 4 byte aligned.

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

#### Display

The default print picture of SQL INTEGER fields is:

PIC "-(10)9"

Family                      Numeric, binary integer

Technical                      The byte layout of the SQL `INTEGER` data type is as follows:

Byte 1	s	b <sub>0</sub>	b <sub>1</sub>	b <sub>2</sub>	b <sub>3</sub>	b <sub>4</sub>	b <sub>5</sub>	b <sub>6</sub>
Byte 2	b <sub>7</sub>	b <sub>8</sub>	b <sub>9</sub>	b <sub>10</sub>	b <sub>11</sub>	b <sub>12</sub>	b <sub>13</sub>	b <sub>14</sub>
Byte 3	b <sub>15</sub>	b <sub>16</sub>	b <sub>17</sub>	b <sub>18</sub>	b <sub>19</sub>	b <sub>20</sub>	b <sub>21</sub>	b <sub>22</sub>
Byte 4	b <sub>23</sub>	b <sub>24</sub>	b <sub>25</sub>	b <sub>26</sub>	b <sub>27</sub>	b <sub>28</sub>	b <sub>29</sub>	b <sub>30</sub>

Size: 4 bytes

s =      sign bit, 1 = negative

b<sub>n</sub> =    b<sub>0</sub> is the most significant bit and b<sub>30</sub> is the least significant bit

Range                          The range of `INTEGER` items is:

Maximum:                  +2147483647

Minimum:                  -2147483648

Examples                      `DEFINE INT : SQL INTEGER`

Defines `INT` as a 4 byte binary integer.

`SQL LONG VARBINARY`      The `SQL LONG VARBINARY` data type represents variable length binary data.

Syntax                          `SQL LONG VARBINARY`

The absolute maximum number of bytes that can be represented with the `SQL LONG VARBINARY` data type is 2,147,483,643. In practice, this limit can probably never be reached due to practical considerations.

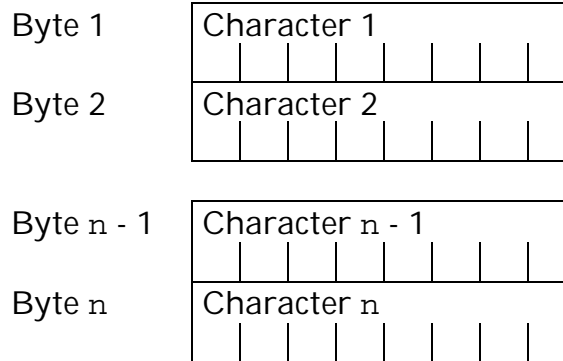
`ALLOW NULLS` may be appended to the data type specification to allow storage of a null value.

Display                          `SQL LONG VARBINARY` fields are printed in hexadecimal using a default print width of `2 * (length + 1)`.

Family                          Binary, variable length

## Technical

The byte layout of SQL LONG VARBINARY items is as follows:



Size: n bytes, where n is the number of bytes required to store a particular item.

## Examples

```
DEFINE BIN : SQL LONG VARBINARY
```

Defines BIN as a long variable length binary string.

## SQL LONG VARCHAR

The SQL LONG VARCHAR data type represents variable length character data.

## Syntax

```
SQL LONG VARCHAR
```

The absolute maximum number of bytes that can be represented with the SQL LONG VARCHAR data type is 2,147,483,643. In practice however, this limit can probably never be reached due to practical considerations.

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

## Display

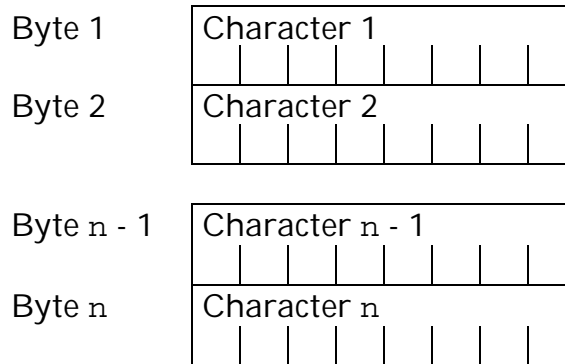
There is no default print width for SQL LONG VARCHAR fields. By default the entire field is printed.

## Family

Character string, variable length

## Technical

The byte layout of SQL LONG VARCHAR items is as follows:



Size: n bytes, where n is the number of bytes required to store a particular item.

### Examples

```
DEFINE CH : SQL LONG VARCHAR
```

Defines CH as a long variable length character string.

### SQL REAL

The `SQL REAL` data type represents a 4 byte IEEE floating point number.

### Syntax

```
SQL REAL
```

On some machines Warehouse uses the type `SQL REAL_` instead of `SQL REAL`. The difference between the two is the data alignment. `SQL REAL` fields may be 1 byte aligned. `SQL REAL_` fields are 4 byte aligned.

`ALLOW NULLS` may be appended to the data type specification to allow storage of a null value.

### Display

The default print picture of `SQL REAL` fields is:

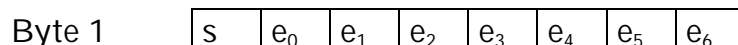
```
PIC "-(9)9.9(6)"
```

### Family

Numeric, floating point

### Technical

The byte layout of the `SQL REAL` data type is as follows:



Byte 2	$e_7$	$f_0$	$f_1$	$f_2$	$f_3$	$f_4$	$f_5$	$f_6$
Byte 3	$f_7$	$f_8$	$f_9$	$f_{10}$	$f_{11}$	$f_{12}$	$f_{13}$	$f_{14}$
Byte 4	$f_{15}$	$f_{16}$	$f_{17}$	$f_{18}$	$f_{19}$	$f_{20}$	$f_{21}$	$f_{22}$

Size: 4 bytes

s = sign bit, 1 = negative

e = exponent, 8 bits, biased by 127

f = fraction, 23 bits, implicit leading 1

Value:  $(-1)^s * 2^{(e - 127)} * (1.f)$

Notes: Exponents of all 0's and all 1's are reserved.

An exponent with all 0's and a fraction of all 0's represents the number 0. An exponent of all 0's and a non-zero fraction represents a denormalized number. An exponent of all 1's and a fraction of all 0's represents infinity. An exponent of all 1's and a non-zero fraction represents a NaN (not a number).

Range

The range of SQL REAL items is:

Maximum:  $+3.402823 \cdot 10^{38}$

Minimum:  $-3.402823 \cdot 10^{38}$

Minimum > 0:  $+1.175495 \cdot 10^{-38}$

Examples

DEFINE FLT : SQL REAL

Defines FLT as an IEEE 4 byte floating point number.

SQL SMALLINT

The SQL SMALLINT data type represents a 2 byte binary signed integer in twos-complement form.

Syntax

SQL SMALLINT

On some machines Warehouse uses the type SQL SMALLINT\_ instead of SQL SMALLINT. The difference between the two is the data alignment. SQL SMALLINT fields may be 1 byte aligned. SQL SMALLINT\_ fields are 2 byte aligned.

ALLOW NULLS may be appended to the data type

specification to allow storage of a null value.

## Display

The default print picture of SQL SMALLINT fields is:

```
PIC "-(5)9"
```

## Family

Numeric, binary integer

## Technical

The byte layout of the SQL SMALLINT data type is as follows:

Byte 1	s	b <sub>0</sub>	b <sub>1</sub>	b <sub>2</sub>	b <sub>3</sub>	b <sub>4</sub>	b <sub>5</sub>	b <sub>6</sub>
Byte 2	b <sub>7</sub>	b <sub>8</sub>	b <sub>9</sub>	b <sub>10</sub>	b <sub>11</sub>	b <sub>12</sub>	b <sub>13</sub>	b <sub>14</sub>

Size: 2 bytes

s = sign bit, 1 = negative

b<sub>n</sub> = b<sub>0</sub> is the most significant bit and b<sub>14</sub> is the least significant bit.

## Range

The range of SQL SMALLINT items is:

Maximum: +32767

Minimum: -32768

## Examples

```
DEFINE INT : SQL SMALLINT
```

Defines INT as a 2 byte binary integer.

## SQL TIME

The SQL TIME data type represents a date and/or time.

## Syntax

```
SQL TIME
```

On some machines Warehouse uses the type SQL TIME\_ instead of SQL TIME. The difference between the two is the data alignment. SQL TIME fields may be 1 byte aligned. SQL TIME\_ fields are 2 byte aligned.

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

**Display** The default print width of SQL TIME fields is 8 in the following format:  
hh:mm:ss

**Family** Date/time

**Technical** The byte layout of SQL TIME items is as follows:

Byte 1	Hour filler (zero)
Byte 2	Hour
Byte 3	Minute filler (zero)
Byte 4	Minute
Byte 5	Second filler (zero)
Byte 6	Second

Size: 6 bytes

**Range** The range of SQL TIME items is:

Maximum: 23:59:59  
Minimum: 00:00:00

**Examples** DEFINE START\_TIME : SQL TIME

Defines START\_TIME as a 16 byte time in SQL format.

SQL TIMESTAMP

The SQL TIMESTAMP data type represents a date and/or time.

**Syntax**

SQL TIMESTAMP

On some machines Warehouse uses the type SQL TIMESTAMP\_ instead of SQL TIMESTAMP. The difference between the two is the data alignment. SQL TIMESTAMP fields may be 1 byte aligned. SQL TIMESTAMP\_ fields are 4 byte aligned.



ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

**Display** The default print width of SQL TIMESTAMP fields is 20 in the following format:  
dd-mmm-yyyy hh:mm:ss

**Family** Date/time

**Technical** The byte layout of SQL TIMESTAMP items is as follows:

Byte 1	Year high bits
Byte 2	Year low bits
Byte 3	Month filler (zero)
Byte 4	Month
Byte 5	Day filler (zero)
Byte 6	Day
Byte 7	Hour filler (zero)
Byte 8	Hour
Byte 9	Minute filler (zero)
Byte 10	Minute
Byte 11	Second filler (zero)
Byte 12	Second
Byte 13	Nanosecond 1 (High)
Byte 14	Nanosecond 2
Byte 15	Nanosecond 3
Byte 16	Nanosecond 4 (Low)



Size: 16 bytes

#### Range

The range of SQL TIMESTAMP items is:

Maximum: December 31, 9999 AD 23:59

Minimum: January 1, 6000 BC 00:00

#### Examples

```
DEFINE ORD_DATE : SQL TIMESTAMP
```

Defines ORD\_DATE as a 16 byte date in SQL format.

### SQL VARBINARY

The SQL VARBINARY data type represents variable length binary data.

#### Syntax

```
SQL VARBINARY(n)
```

*n* specifies the maximum length in bytes of the field. *n* must be from 1 to 254.

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

#### Display

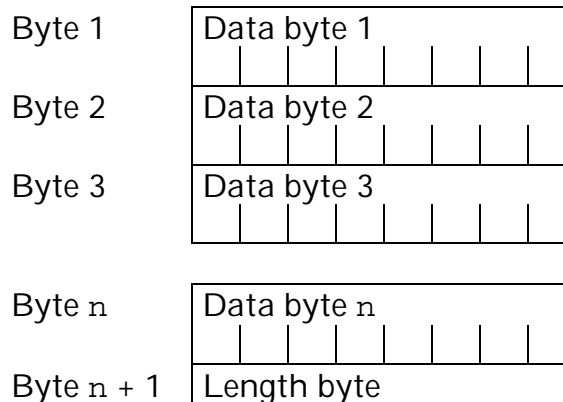
SQL VARBINARY printed in hexadecimal using a default print width of  $2 * (n + 1)$ .

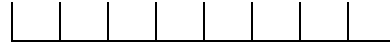
#### Family

Binary, variable length

#### Technical

The byte layout of SQL VARBINARY items is as follows:





Size:  $n + 1$  bytes

### Examples

```
DEFINE BIN : SQL VARBINARY(20)
DEFINE BIGBIN : SQL VARBINARY(200)
```

Defines BIN as a variable length binary string capable of holding up to 20 bytes. Defines BIGBIN as a variable length binary string capable of holding up to 200 bytes.

### SQL VARCHAR

The SQL VARCHAR data type represents variable length character data.

### Syntax

```
SQL VARCHAR(n)
```

$n$  specifies the maximum length in bytes of the field.  $n$  must be from 1 to 3996.

ALLOW NULLS may be appended to the data type specification to allow storage of a null value.

### Display

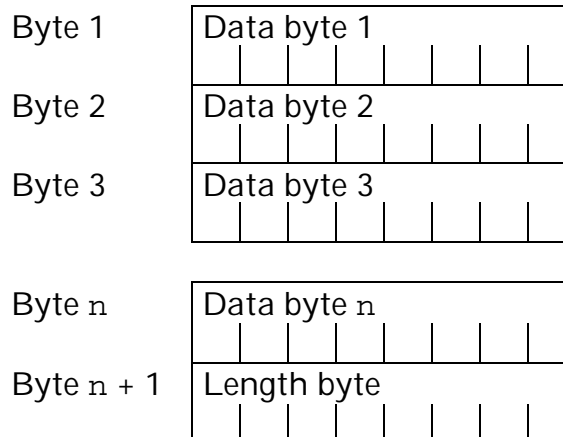
The default print width of SQL VARCHAR fields is the width of the field, or  $n$ .

### Family

Character string, variable length

### Technical

The byte layout of SQL VARCHAR items is as follows:



Size:  $n + 1$  bytes

Examples

```
DEFINE CH : SQL VARCHAR(20)
DEFINE BIGCH : SQL VARCHAR(2000)
```

Defines CH as a variable length character string capable of holding up to 20 characters. Defines BIGCH as a variable length character string capable of holding up to 2000 characters.

**Warehouse Data Types**

The following data types are Warehouse data types. Warehouse data types are used internally within Warehouse for many operations.

Supported Warehouse data types are as follows:

ARRAY	Array of items.
BINARY	Variable length binary data
BOOLEAN	Logical (true/false) data
[TEXT] CHAR	Fixed length character data
CHRONOS	Six byte date/time
DATE	A calendar date
DATETIME	A calendar date and time
FLOAT	Floating point numeric
INTEGER	Variable length integer
INTERVAL	Difference between dates/times
NSTRING	Variable length native (double-byte) character data
NUMERIC	Variable length numeric
RECORD	Data records
[TEXT] SIGNED	Fixed length signed numeric
STRING	Variable length character data
TIME	A time within day
[TEXT] UNSIGNED	Fixed length unsigned numeric

TEXT types are optional for CHAR, SIGNED, and UNSIGNED. This has no effect on the data type and is provided for readability.

**Null Values**

Warehouse data types can be designated as allowing nulls by specifying `ALLOW NULLS` after the type name, e.g. `INTEGER ALLOW NULLS`.

The following is a detailed description of each Warehouse data type.

**ARRAY**

The ARRAY data type represents an array of items.

**Syntax**

`ARRAY [lo-ix .. hi-ix] OF data-type`

`lo-ix` is the minimum index of the array and must

be less than or equal to `hi-ix`.

`hi-ix` is the maximum index of the array and must be greater than or equal to `lo-ix`.

`data-type` is any Warehouse type specification including records and arrays.

#### Display

Arrays are printed by field with each field separated by one space.

#### Family

Array

#### Technical

The byte layout of arrays depends on the fields within the record. The alignment of each field is adjacent to the previous field with no filler. The exception to this is Warehouse data types, which may be 4 byte aligned.

Elements of an array are accessed using square brackets around the array index as follows:

```
array-name [ index ]
```

The number of elements in the array is equal to:  
 $(\text{hi-index} - \text{lo-index}) + 1$

#### Examples

##### Example 1

```
DEFINE MONTHLY_TOTALS : &
    ARRAY [1..12] OF ORACLE NUMBER
DEFINE GRAND_TOTAL : ORACLE NUMBER
DEFINE IX : ORACLE NUMBER

SETVAR GRAND_TOTAL = 0
SETVAR IX = 1
WHILE IX <= 12
    SETVAR GRAND_TOTAL = &
        GRAND_TOTAL + MONTHLY_TOTALS[IX]
    SETVAR IX = IX + 1
ENDWHILE
```

Defines `MONTHLY_TOTALS` as an array of Oracle numbers. The variables `GRAND_TOTAL` and `IX` are also defined as Oracle numbers. A `WHILE` loop is then used to add up each element of `MONTHLY_TOTALS` and put the sum into

```
GRAND_TOTAL.
```

### Example 2

```
FORMAT ORD-FMT : RECORD
  CUST-NO       : X6
  ORDER-NO      : X10
  ORD-DATE      : X6
  ORD-AMT       : Z10
END
DEFINE ORDS : &
  ARRAY [1..1000] OF FORMAT ORD-FMT
```

Defines ORDS as an array of 1000 order records.

## BINARY

The BINARY data type represents variable length binary data.

### Syntax

```
BINARY
```

### Display

BINARY fields may not be printed.

### Family

Binary

### Technical

The byte layout of BINARY items is internal to Warehouse.

### Example

```
DEFINE BIN : BINARY
```

Defines BIN as a variable length binary field.

## BOOLEAN

The BOOLEAN data type represents Boolean, or true/false data.

### Syntax

```
BOOLEAN
```

### Display

The default print width of BOOLEAN fields is 6.

### Family

Logical

### Technical

The byte layout of BOOLEAN items is internal to Warehouse.

### Range

A BOOLEAN item may only represent \$TRUE or

	<pre>\$FALSE.</pre>
Example	<pre>DEFINE MYFLAG : BOOLEAN SETVAR MYFLAG = \$TRUE</pre> <p>Defines MYFLAG as a Boolean field, then sets the value of MYFLAG to \$TRUE.</p>
CHAR	<p>The CHAR data type represents fixed length character data.</p>
Syntax	<pre>CHAR(n)</pre> <p>n specifies the length in bytes of the field.</p>
Display	<p>The default print width of CHAR fields is the width of the field, or n.</p>
Family	<p>Character string, fixed length</p>
Technical	<p>The byte layout of CHAR items is as follows:</p> <div><div>Byte 1</div><div>Character 1</div><div>Byte 2</div><div>Character 2</div><div>Byte n</div><div>Character n</div></div> <p>Size: n bytes</p>
Examples	<pre>DEFINE CH : CHAR(20) DEFINE BIGCH : CHAR(2000)</pre> <p>Defines CH as a 20 byte fixed length character string. Defines BIGCH as a 2000 byte fixed length character string.</p>
CHONOS	<p>The CHONOS data type represents a 6 byte date and time.</p>
Syntax	<pre>CHONOS</pre>



## Display

The default print picture of CHONOS fields is:

```
PIC "DD-MON-YYYY HH24:MI:SS"
```

## Family

Date

## Technical

The byte layout of the CHONOS data type is as follows:

Byte 1	y <sub>0</sub>	y <sub>1</sub>	y <sub>2</sub>	y <sub>3</sub>	y <sub>4</sub>	y <sub>5</sub>	y <sub>6</sub>	y <sub>7</sub>
Byte 2	y <sub>8</sub>	y <sub>9</sub>	y <sub>10</sub>	y <sub>11</sub>	d <sub>0</sub>	d <sub>1</sub>	d <sub>2</sub>	d <sub>3</sub>
Byte 3	d <sub>4</sub>	d <sub>5</sub>	d <sub>6</sub>	d <sub>7</sub>	d <sub>8</sub>	h <sub>0</sub>	h <sub>1</sub>	h <sub>2</sub>
Byte 4	h <sub>3</sub>	h <sub>4</sub>	m <sub>0</sub>	m <sub>1</sub>	m <sub>2</sub>	m <sub>3</sub>	m <sub>4</sub>	m <sub>5</sub>
Byte 5	s <sub>0</sub>	s <sub>1</sub>	s <sub>2</sub>	s <sub>3</sub>	s <sub>4</sub>	s <sub>5</sub>	s <sub>6</sub>	s <sub>7</sub>
Byte 6	s <sub>8</sub>	s <sub>9</sub>	s <sub>10</sub>	s <sub>11</sub>	s <sub>12</sub>	s <sub>13</sub>	s <sub>14</sub>	s <sub>15</sub>

Size: 6 bytes

y = year, 1-4095, 12 bits

d = day within year, 1-366, 9 bits

h = hour within day, 0-23, 5 bits

m = minute within hour, 0-59, 6 bits

s = millisecond within minute, 0-59999, 16 bits

## Range

The range of CHONOS items is:

Maximum: December 31, 4095 AD 23:59

Minimum: January 1, 1 AD 00:00

## Examples

```
DEFINE CHRONDAT : CHRONOS
```

Defines CHRONDAT as a 6 byte date/time.

## DATE

The DATE data type represents a calendar date without a time.

## Syntax

DATE

## Display

The default print picture of DATE fields is:

PIC "DD-MON-YYYY"

Family Date

Technical The byte layout of DATE items is internal to Warehouse.

Range The range of DATE items is:

Maximum: December 31, 9999 AD

Minimum: January 1, 6000 BC

Example DEFINE DAT : DATE

Defines DAT as a date field.

DATETIME The DATETIME data type represents a calendar date and a time within the day.

Syntax DATETIME

Display The default print picture of DATETIME fields is:

PIC "DD-MON-YYYY HH24:MI:SS"

Family Date

Technical The byte layout of DATETIME items is internal to Warehouse.

Range The range of DATETIME items is:

Maximum: December 31, 9999 AD 00:00

Minimum: January 1, 6000 BC 23:59

Example DEFINE DTM : DATETIME

Defines DTM as a date and time field.

FLOAT The FLOAT data type represents floating point data.

Syntax FLOAT

Display                      The default print picture of `FLOAT` fields is:

```
PIC "-(9)9.9(6)"
```

Family                      Numeric, floating point

Technical                    The byte layout of `FLOAT` items is internal to Warehouse.

Range                        A `FLOAT` item may contain virtually any exponent, but performance degrades as the exponent gets larger. The precision of `FLOAT` items is 77 digits of accuracy.

Example                      

```
DEFINE MYFLT : FLOAT
SETVAR MYFLT = -1234.56
```

Defines `MYFLT` as a variable length numeric field and sets the value of `MYFLT` to `-1234.56`.

`INTEGER`                    The `INTEGER` data type represents variable length integer data.

Syntax                      `INTEGER`

Display                      There is no specific default print picture for `INTEGER` fields. By default the entire field is printed.

Family                      Numeric

Technical                    The byte layout of `INTEGER` items is internal to Warehouse.

Range                        Any size of positive and negative integers may be represented with the type `INTEGER`.

Example                      

```
DEFINE MYINT : INTEGER
SETVAR MYINT = -1234
```

Defines `MYINT` as a variable length numeric field and sets the value of `MYINT` to `-1234`.

INTERVAL	The INTERVAL data type represents a difference between two dates and times. An interval is stored as a positive or negative number of days and a positive number of hours.
Syntax	INTERVAL
Display	The default print picture of INTERVAL fields is:  PIC "NNNNNN HH24:MI:SS"
Family	Date
Technical	The byte layout of INTERVAL items is internal to Warehouse.
Example	DEFINE INT : INTERVAL  Defines INT as an interval.
NSTRING	The NSTRING data type represents variable length native (double byte) character data.
Syntax	NSTRING
Display	There is no specific default print width for NSTRING fields. By default the entire field is printed.
Family	Character string, native
Technical	The byte layout of NSTRING fields is internal to Warehouse.  NSTRING items are automatically converted to STRING items where appropriate.
Example	DEFINE MYSTR : NSTRING SETVAR MYSTR = "Taurus Software"  Defines MYSTR as a variable length native character string and sets its value to Taurus Software.

NUMERIC                      The NUMERIC data type represents variable length numeric data.

Syntax                      NUMERIC

Display                      The default print picture of NUMERIC fields is:

PIC "-(9)9.9(6)"

Family                      Numeric

Technical                      The byte layout of NUMERIC items is internal to Warehouse.

Range                      Any size of positive and negative numbers with decimal points may be represented with the type NUMERIC.

Example                      

```
DEFINE MYNUM : NUMERIC
SETVAR MYNUM = -1234.56
```

Defines MYNUM as a variable length numeric field and sets the value of MYNUM to -1234.56.

RECORD                      The RECORD type is used to represent data records containing multiple fields.

Syntax                      Record definitions may only be used in the DEFINE and FORMAT statements. The syntax is as follows:

```
DEFINE var-name : RECORD
    field-1 : type-1 [OFFSET offset-1]
    field-2 : type-2 [OFFSET offset-2]
    .
    .
    .
END
```

or

```
FORMAT fmt-name : RECORD
    field-1 : type-1 [OFFSET offset-1]
    field-2 : type-2 [OFFSET offset-2]
    .
```

```

      .
      .
END

```

When a `RECORD` is specified with the `DEFINE` statement, a variable is created named `var-name` along with the storage for the record.

When a `RECORD` is specified with the `FORMAT` statement, a format specification called `fmt-name` is defined for later use by a `READ` or `DEFINE` statement. No storage is created for the record.

Display	Records are printed by field with each field separated by one space.
Family	Data structure
Technical	<p>The byte layout of records depends on the fields within the record. The alignment of each field is adjacent to the previous field with no filler. The exception to this is Warehouse data types, which may be 4 byte aligned, and the <code>SQL _</code> data types which are aligned depending upon the type.</p> <p>If <code>OFFSET</code> is specified for the field, the field is placed starting at the byte position indicated with offset 1 being the beginning of the record. If no <code>OFFSET</code> is specified, the field is placed at the end of the record. Using <code>OFFSET</code> allows overlapping fields to be created.</p>
Examples	<pre> FORMAT ORD-FMT : RECORD   CUST-NO       : X6   ORDER-NO      : X10   ORD-DATE      : X6   ORD-AMT       : Z10 END DEFINE ORD-REC : FORMAT ORD-FMT </pre> <p>Defines <code>ORD-FMT</code> as the record layout of an order containing 4 fields, and a record width of 32 bytes. The variable <code>ORD-REC</code> is created using the <code>DEFINE</code> statement. 32 bytes of storage are allocated for <code>ORD-REC</code>.</p>

**SIGNED**

The **SIGNED** data type represents fixed point signed numeric data.

**Syntax**

**SIGNED** (*n*)  
or  
**SIGNED** (*n*,*m*)

*n* specifies the maximum number of digits the field may hold, including digits to the right of the decimal point.

If *m* is specified, *m* indicates the number of digits to the right of the decimal point the field may hold. If *m* is not specified, *m* is assumed to be 0. *m* must be from 0 to *n*.

**Display**

The default print picture of **SIGNED** fields is:

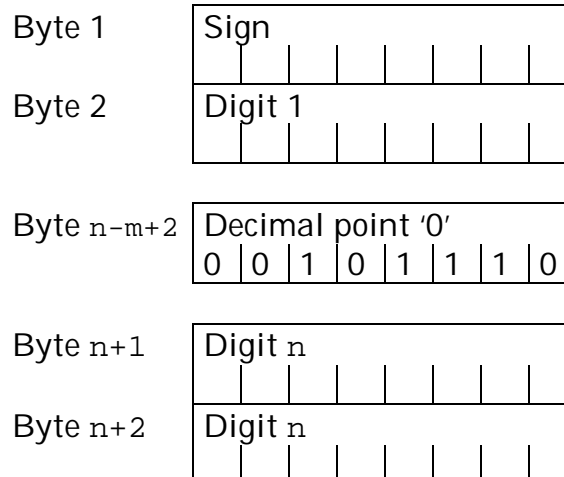
**PIC** "-(*n-m*)9.9(*m*)" for *m* > 0  
**PIC** "-(*n*)9" for *m* = 0

**Family**

Numeric

**Technical**

The byte layout of **SIGNED** type items is as follows when *m* > 0:

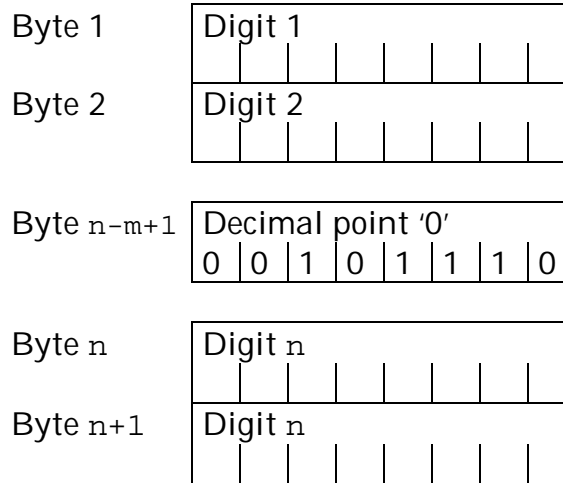


**SIGNED** type items require one byte per digit stored (*n*), plus one byte for the sign, plus one byte for the decimal point.

	<p>Size: <math>n + 2</math> bytes for <math>m &gt; 0</math>  <math>n + 1</math> bytes for <math>m = 0</math></p>
Range	<p>The range of SIGNED items is:</p> <p>Maximum: <math>+99..99</math> with <math>n - m</math> digits  Minimum: <math>-99..99</math> with <math>n - m</math> digits</p>
Examples	<pre>DEFINE DEC : SIGNED (6) DEFINE AMT : SIGNED (10,2)</pre> <p>Defines DEC as a 7 byte fixed decimal integer that can hold up to 6 digits. The default print picture for DEC is "<math>-(6)9</math>". Defines AMT as a 12 byte fixed decimal integer that can hold up to 10 digits: 8 digits to the left of the decimal point, and 2 digits to the right of the decimal point. The default print picture for AMT is "<math>-(8)9.9(2)</math>"</p>
STRING	<p>The STRING data type represents variable length character data.</p>
Syntax	STRING
Display	There is no specific default print width for STRING fields. By default the entire field is printed.
Family	Character string
Technical	The byte layout of STRING fields is internal to Warehouse.
Example	<pre>DEFINE MYSTR : STRING SETVAR MYSTR = "Taurus Software"</pre> <p>Defines MYSTR as a variable length character field and sets its value to Taurus Software.</p>
TIME	<p>The TIME data type represents a time within the day without an associated date.</p>
Syntax	TIME



Display	<p>The default print picture of DATETIME fields is:</p> <pre>PIC "HH24:MI:SS"</pre>
Family	Date
Technical	The byte layout of TIME items is internal to Warehouse.
Range	<p>The range of TIME items is:</p> <p>Maximum: 23:59:59 Minimum: 00:00:00</p>
Example	<pre>DEFINE TIM : TIME</pre> <p>Defines TIM as a date field.</p>
UNSIGNED	The UNSIGNED data type represents fixed point unsigned numeric data.
Syntax	<pre>UNSIGNED (n)</pre> <p>or</p> <pre>UNSIGNED (n,m)</pre> <p>n specifies the maximum number of digits the field may hold, including digits to the right of the decimal point.</p> <p>If m is specified, m indicates the number of digits to the right of the decimal point the field may hold. If m is not specified, m is assumed to be 0. m must be from 0 to n.</p>
Display	<p>The default print picture of UNSIGNED fields is:</p> <pre>PIC "(n-m)9.9(m)" for m &gt; 0 PIC "(n)9" for m = 0</pre>
Family	Numeric
Technical	The byte layout of UNSIGNED type items is as follows when m > 0:



UNSIGNED type items require one byte per digit stored (n) plus one byte for the decimal point.

Size:  $n + 1$  bytes for  $m > 0$

$n$  bytes for  $m = 0$

#### Range

The range of UNSIGNED items is:

Maximum:  $+99..99$  with  $n - m$  digits

Minimum: 0

#### Examples

```
DEFINE DEC : UNSIGNED (6)
DEFINE AMT : UNSIGNED (10,2)
```

Defines DEC as a 6 byte fixed unsigned integer that can hold up to 6 digits. The default print picture for DEC is "(6)9". Defines AMT as a 11 byte fixed unsigned integer that can hold up to 10 digits: 8 digits to the left of the decimal point, and 2 digits to the right of the decimal point. The default print picture for AMT is "(8)9.9(2)"

## **Chapter Seven**

### **Installation and Execution**

Warehouse is installed and run differently on each supported platform. This chapter contains a section on how the installation is performed, how to run Warehouse, and how the Warehouse server program is set up for each of the supported platforms.

**Warehouse Client**

The Warehouse client program is the program that is run by users and processes Warehouse scripts. When the Warehouse client program is run it may be given options and a script file name on the command line. When a script file name is on the command line Warehouse processes the script file like the `XEQ` command.

**Client Options**

The Warehouse client program may be run with the following command line options (case insensitive):

- `-connect` Causes Warehouse to enter a special mode for checking Warehouse server connections. `-connect` may be abbreviated as `-c`.
- `-nostats` Causes statistics to be suppressed after the script has completed. Using `-nostats` in the command line has the same effect as putting `SET STATS OFF` in the script.
- `-pause` Causes Warehouse to prompt the user to press ENTER before exiting. This can be useful in a Windows environment where the window is closed after Warehouse exits. `-pause` may be abbreviated as `-p`.
- `-showversion` Displays the version number of the program and exits. This option can be used to verify the Warehouse client program can be run.
- `-start` Causes the Warehouse banner to be suppressed and when run with a script file name, the script is processed without displaying the lines just like

the START statement.

- validate Causes Warehouse to display a menu for validating Warehouse. This is typically done by the system administrator as part of an installation or upgrade. -validate may be abbreviated as -v.
  
- showinfo Displays the Warehouse operating environment. When either the Warehouse client or server is run with -showinfo, the operating environment is displayed. The syntax is :

```
warehouse -showinfo [output-file]
```

If output-file is specified, the information is written to the specified file.

The -showinfo information looks like this:

```
Program version      : Warehouse 3.00.4377-W
Program name         :
                     d:\wh\current\whide\wh.exe
Warehouse home       : d:\wh\current\whide\
Operating system     : MS Windows
System ID            : 000cfl8eda7
Validation status    : Production
System name          : CAESAR
User login name      : Administrator
Word size            : 32 bits (Little
                     endian)
Current date and time : 10-OCT-2004
                     08:23:52
File system          : Static
Posix libraries      : Unavailable
TurboIMAGE           : Unavailable
ODBC libraries       : Available
Oracle               : Available (Dynamic)
Oracle 8 features    : Available (Dynamic)
```

Program version - This is version of the Warehouse program.

Program name - This is the name of Warehouse program.

Warehouse client program - Indicates the name of the Warehouse client program. Only the Warehouse server displays this value. It is necessary for the server to be able to find the client program when using DataBridger Studio job control features.

Warehouse program type - Indicates the type of the Warehouse program. This value is "Server" for the Warehouse server program and is not displayed for the Warehouse client program.

Warehouse home - This is the directory where Warehouse looks for its files. This includes the validation file and configuration files such as CHARMAPS.

Operating system - This is operating system under which the Warehouse program was compiled. It corresponds to the last character of the Warehouse version as follows:

- A - DEC Alpha (No longer supported)
- C - SCO (No longer supported)
- F - HP-UX PA RISC 2.0 (No longer supported)
- G - HP-UX Itanium
- H - HP-UX PA RISC 1.1
- I - SGI IRIX (No longer supported)
- J - SunOS 5.10 Solaris X86
- L - Red Hat Linux
- M - MPE/iX
- Q - AS/400
- R - IBM RS/6000 AIX
- S - SunOS/Solaris SPARC
- T - Debian Linux
- U - Microsoft Windows x64
- V - DEC VMS (No longer supported)
- W - MS Windows
- X - Unknown

System ID - This is the unique system ID for the system on which Warehouse is running.

Validation status - Shows the status of the Warehouse validation as follows:

`Production` Indicates Warehouse has been validated for production.  
`Expires on <date>` Indicates a demonstration version that expires on the date shown.  
`Expired on <date>` Indicates an expired demonstration version that expired on the date shown.  
`Needs validation` Indicates that Warehouse needs to be validated before it can be run.

`System name` - This is the system name of the system on which Warehouse is running. This name is determined by the system network configuration. This name may be important when setting up the `AUTHFILE` for Warehouse server access.

`User login name` - This is the name of the login user as indicated by the system. This name may be important when setting up the `AUTHFILE` for Warehouse server access.

`Word size` - indicates the hardware environment under which the Warehouse program was compiled. Currently all supported platforms are 32 bits. Big endian indicates forward byte ordering (e.g. 0x01020304 is stored as 0x01 0x02 0x03 0x04) and Little endian indicates backward byte ordering (e.g. 0x01020304 is stored as 0x04 0x03 0x02 0x01)

`Current date and time` - Indicates the system clock time as determined by the C library. This may be different than other methods of determining the system time, particularly on MPE/iX.

`OS Date conflicts` - Indicates the system clock as determined by operating system calls conflicts with the C library clock. This time is only displayed if there is a conflict with C library date and time.



This value is often displayed on MPE/iX when the TZ environment variable is not set or is set incorrectly.

File system - Indicates which file system is used by the Warehouse program. This is always "Static" except on HP-UX platforms. The values are:

Static Indicates the standard file system.

Static (Dyn load err) Indicates the standard file system after an attempt to dynamically load the HP-UX 64-bit file system failed. (Only used on HP-UX)

Dynamic Indicates the standard file system was dynamically loaded.

Dynamic 64 bit Indicates the HP-UX 64-bit file system was dynamically loaded. (Only used on HP-UX)

Posix libraries - Indicates whether the Posix libraries are used. The value of the does not matter, except on MPE/iX. On MPE/iX, it is possible some file operations may generate different results if the Posix libraries are used. On MPE/iX, both Posix and non-Posix versions of Warehouse are shipped.

HP-UX Trusted system - (Only on HP-UX)  
Indicates if the system is considered a trusted system. Trusted systems use a different method for password verification when a client connects to the Warehouse server. A trusted system is indicated by the presence of the file  
/tcb/files/auth/system/default

TurboIMAGE - Indicates whether IMAGE is available or not. Always "Available" on MPE/iX, and always "Unavailable" on other platforms. This may change some day with the implementation of Eloquence.

ODBC libraries - Indicates whether the ODBC are available or not. Always "Available" on Windows, and always "Unavailable" on other platforms

except for IBM RS/6000. On the IBM RS/6000, "Available" indicates that DB2 may be accessed and "Unavailable" indicates that DB2 may not be accessed.

Oracle - Indicates whether the Oracle is available and the method used to link the Oracle libraries. Values are:

- Unavailable Indicates Oracle is not available with this Warehouse program.
- Unavailable (Dynamic) Indicates Oracle is not available with this Warehouse program after an attempt was made to dynamically load the Oracle OCI calls. This may be able to be corrected by setting environment variables such as ORACLE\_HOME.
- Available (Static) Indicates Oracle is available with this Warehouse program and the Oracle OCI libraries were linked at compile time.
- Available (Dynamic) Indicates Oracle is available with this Warehouse program and the Oracle OCI libraries were linked at run time.

Oracle 8 features - Indicates whether the Oracle 8.1 OCI features are available. These features are necessary for accessing CLOB and BLOB data in tables. These features are always dynamically loaded.

- Unavailable (Dynamic) Indicates Oracle 8.1 OCI features for CLOB/BLOB access are not available after an attempt was made to dynamically load the Oracle OCI calls. It may be possible to correct this by setting environment variables such as WHORACLE8LIB.
- Available (Dynamic) Indicates Oracle 8.1 OCI features for CLOB/BLOB access are available after linking them at run time.

**Examples****Example 1**

```
/usr/local/taurus/whii/warehouse myscript
```

This example runs the Warehouse client on Unix or an AS400 which processes the Warehouse script file `myscript` like the `XEQ` command.

**Example 2**

```
RUN WH.WHII.TAURUS;INFO="-NOSTATS -START SCRIPT2"
```

This example runs the Warehouse client on MPE/iX using the `-nostats` and `-start` options which suppress the Warehouse banner, statistics and display of the script file `SCRIPT2` as it is processed.

**Example 3**

```
C:\Program Files\Taurus\Warehouse\wh -pause
```

This example runs the Warehouse client on Windows using the `-pause` option causing Warehouse to prompt the user to press ENTER before Warehouse exits.

**Warehouse Server**

The Warehouse server program is a program that runs in the background and receives connections from Warehouse clients when they connect to a `REMOTE` database. The server program must run with privileges that allow it to change the login to the user connection. For security reasons a file called the `AUTHFILE` is set up by the system administrator and is used by the server to determine how connections can be made.

**Process ID**

For non-Windows systems, the process id (pid) of the main Warehouse server process is written to a file in the same directory in which the server program resides. The pid file is called `WHSPID`. Signals can be sent to Warehouse server processes by logging in as root (superuser) and using the `kill` command with the process as a parameter:

```
Kill 'cat /home/taurus/whii/WHSPID'
```

If the main Warehouse server process receives a `SIGTERM` signal (the default for the `kill` command), all child processes are sent a `SIGTERM` and then the main server process exits.

If a child Warehouse server process (i.e. one that is processing requests from a Warehouse script or from DataBridger Studio) receives a `SIGTERM` signal, `WHERR 20035` is sent to the client program and then the server process exits.

If the main Warehouse server process receives a `SIGUSR1` signal (i.e. `kill -SIGUSR1`), a list of child processes is written to the file `WHPROCS` in the same directory as the server program.

#### Server Options

The Warehouse server program may be run with the following command line options (case insensitive):

`-authfile` Causes Warehouse to display a menu for editing the `AUTHFILE`. (The Warehouse client also accepts this option, but may not be able to access or write to the `AUTHFILE`.) `-authfile` may be abbreviated as `-a`.

`-port n` Causes the Warehouse server to receive connections on port `n` instead of the usual Warehouse port 1610. For a Warehouse client to connect to a port other than the default port (1610), the port number must be specified after the system name using a colon separator in the `OPEN` statement. Example:

```
OPEN DB REMOTE &  
      REMSYS:32400 &  
      USER=RUSER PASS=RPASS &  
      ORACLE SCOTT/TIGER
```

`-serverinfo` used to display the Warehouse operating environment for a remote Warehouse server. The syntax is:

```
warehouse -serverinfo [server-name]
```

If server-name is specified, information is displayed for the server. If server-name is not specified, the user is prompted for the server name. The information displayed is the same as the `-showinfo` information documented below with the addition of the following information:

```
Warehouse program type : Server
Server IP Address   : #.#.#.#
                    (server.taurus.com)
Client IP Address   : #.#.#.#
                    (client.taurus.com)
```

This option also shows job control databases on the server.

`-showversion` Displays the version number of the program and exits. This option can be used to verify the Warehouse server program can be run.

`-showinfo` Displays the Warehouse operating environment. When either the Warehouse client or server is run with `-showinfo`, the operating environment is displayed. The syntax is:

```
warehouse -showinfo [output-file]
```

If output-file is specified, the information is written to the specified file.

The -showinfo information looks like this:

```
Program version      :Warehouse 3.00.4377-W
Program name        :
                    d:\wh\current\whide\wh
                    .exe
Warehouse client program
                    : C:\Program
                    Files\TAURUS\Warehouse
                    \WH.EXE
Warehouse home       : d:\wh\current\whide\
Operating system     : MS Windows
System ID            : 000cf1d8eda7
Validation status    : Production
System name          : CAESAR
User login name      : Administrator
Word size            : 32 bits (Little
                    endian)
Current date and time
                    : 10-OCT-2004 08:23:52
File system          : Static
Posix libraries      : Unavailable
TurboIMAGE           : Unavailable
ODBC libraries       : Available
Oracle               : Available (Dynamic)
Oracle 8 features    : Available (Dynamic)
Job control databases
                    : None
```

Program version - This is version of the Warehouse program.

Program name - This is the name of Warehouse program.

Warehouse client program - Indicates the name of the Warehouse client program. Only the Warehouse server displays this value. It is necessary for the server to be able to find the client program when using DataBridger Studio job control features.

Warehouse program type - Indicates the type of the Warehouse program. This value is

"Server" for the Warehouse server program and is not displayed for the Warehouse client program.

Warehouse home - This is the directory where Warehouse looks for its files. This includes the validation file and configuration files such as CHARMAPS.

Operating system - This is operating system under which the Warehouse program was compiled. It corresponds to the last character of the Warehouse version as follows:

- A - DEC Alpha (No longer supported)
- C - SCO (No longer supported)
- F - HP-UX 11
- H - HP-UX
- I - SGI Irix
- L - Linux
- M - MPE/iX
- R - IBM RS/6000
- S - SunOS
- V - DEC VMS (No longer supported)
- W - MS Windows
- X - Unknown

System ID - This is the unique system ID for the system on which Warehouse is running.

Validation status - Shows the status of the Warehouse validation as follows:

Production Indicates  
Warehouse has been validated  
for production.  
Expires on <date> Indicates a  
demonstration version that

expires on the date shown.

Expired on <date> Indicates an expired demonstration version that expired on the date shown.

Needs validation Indicates that Warehouse needs to be validated before it can be run.

System name - This is the system name of the system on which Warehouse is running. This name is determined by the system network configuration. This name may be important when setting up the AUTHFILE for Warehouse server access.

User login name - This is the name of the login user as indicated by the system. This name may be important when setting up the AUTHFILE for Warehouse server access.

Word size - indicates the hardware environment under which the Warehouse program was compiled. Currently all supported platforms are 32 bits. Big endian indicates forward byte ordering (e.g. 0x01020304 is stored as 0x01 0x02 0x03 0x04) and Little endian indicates backward byte ordering (e.g. 0x01020304 is stored as 0x04 0x03 0x02 0x01)

Current date and time - Indicates the system clock time as determined by the C library. This may be different than other methods of determining the system time, particularly on MPE/iX.



`OS Date conflicts` - Indicates the system clock as determined by operating system calls conflicts with the C library clock. This time is only displayed if there is a conflict with C library date and time. This value is often displayed on MPE/iX when the TZ environment variable is not set or is set incorrectly.

`File system` - Indicates which file system is used by the Warehouse program. This is always "Static" except on HP-UX platforms. The values are:

`Static` Indicates the standard file system.

`Static (Dyn load err)`

Indicates the standard file system after an attempt to dynamically load the HP-UX 64-bit file system failed. (Only used on HP-UX)

`Dynamic` Indicates the standard file system was dynamically loaded.

`Dynamic 64 bit` Indicates the HP-UX 64-bit file system was dynamically loaded. (Only used on HP-UX)

`Posix libraries` - Indicates whether the Posix libraries are used. The value of the does not matter, except on MPE/iX. On MPE/iX, it is possible some file operations may generate different results if the Posix libraries are used. On MPE/iX, both Posix and non-Posix versions of Warehouse are shipped.

HP-UX Trusted system - (Only on HP-UX) Indicates if the system is considered a trusted system. Trusted systems use a different method for password verification when a client connects to the Warehouse server. A trusted system is indicated by the presence of the file  
`/tcb/files/auth/system/default`

TurboIMAGE - Indicates whether IMAGE is available or not. Always "Available" on MPE/iX, and always "Unavailable" on other platforms. This may change some day with the implementation of Eloquence.

ODBC libraries - Indicates whether the ODBC are available or not. Always "Available" on Windows, and always "Unavailable" on other platforms except for IBM RS/6000. On the IBM RS/6000, "Available" indicates that DB2 may be accessed and "Unavailable" indicates that DB2 may not be accessed.

Oracle - Indicates whether the Oracle is available and the method used to link the Oracle libraries. Values are:

Unavailable Indicates Oracle is not available with this Warehouse program.

Unavailable (Dynamic)

Indicates Oracle is not available with this Warehouse program after an attempt was made to dynamically load the Oracle OCI calls. This may be able to

be corrected by setting environment variables such as ORACLE\_HOME.

Available (Static) Indicates Oracle is available with this Warehouse program and the Oracle OCI libraries were linked at compile time.

Available (Dynamic) Indicates Oracle is available with this Warehouse program and the Oracle OCI libraries were linked at run time.

Oracle 8 features - Indicates whether the Oracle 8.1 OCI features are available. These features are necessary for accessing CLOB and BLOB data in tables. These features are always dynamically loaded.

Unavailable (Dynamic) Indicates Oracle8.1 OCI features for CLOB/BLOB access are not available after an attempt was made to dynamically load the Oracle OCI calls. It may be possible to correct this by setting environment variables such as WHORACLE8LIB.

Available (Dynamic) Indicates Oracle 8.1 OCI features for CLOB/BLOB access are available after linking them at run time.

Job Control databases - Shows the contents of the job control database WHCTLDBS that were set up through DataBridger Studio.

Examples

Example 1

```
/usr/local/taurus/whii/whserv -port 32400 &
```

This example runs the Warehouse server as a background process on Unix using the `-port` option causing Warehouse to listen for client connections on port 32400.

## **INI File**

The Warehouse `.ini` file contains initial values for Warehouse `SET` options. On entry, Warehouse searches for a file called `wh.ini` or `WHINI` in the same directory as the Warehouse program. If a `.ini` file is found, it is opened and the options are processed. The file layout is similar to `.ini` files in the Microsoft Windows environment. The first line must contain `[Warehouse]` and subsequent lines are used to set initial values for global options. Options are set using the syntax:

```
option-name = option-value
```

Options available within the Warehouse `.ini` file are: `AUTOPAD`, `MSGs`, `PAGELLENGTH`, `PAGEWIDTH`, `START`, and `STATs`. See the `SET` statement for information on option details. Options in the `.ini` file may be overridden within a script using the `SET` statement.

Example `WH.INI` file:

```
[Warehouse]
AUTOPAD=ON
PAGELLENGTH=59
```

The above Warehouse `.ini` file sets the `AUTOPAD` option to `ON` and the `PAGELLENGTH` to 59 when the Warehouse client is run.

**Installation on MPE/iX**

On MPE/iX Warehouse is installed by the installation program developed by Taurus Software called EZ-Install/3000. All Warehouse files are installed into the WHII group of the TAURUS account. If the MPE/iX system is not attached to a network, then tape media will be required. To install Warehouse on MPE/iX, proceed as follows:

**Web download**

1. Start Reflection on PC where the self-extracting file named whii.wrq is located.
2. Logon to the target MPE/iX as `MANAGER.SYS,PUB`
3. Using Reflection, transfer `whii.wrq` with the Transfer Type attribute set to Labels.
4. Run the resulting WHII program: `RUN WHII`
5. The program WHII automatically builds the account TAURUS. When prompted for the TAURUS account password enter an account password of your choosing (it will not be displayed), and enter the same password again for verification, when prompted.
6. When prompted for your Warehouse validation code, enter the demonstration validation code provided by your Taurus Support representative.
7. The WHII program then proceeds to restore all Warehouse files. This may take a few minutes. Once the files have been restored, your Warehouse installation will be complete.

**Tape Install**

1. Logon as `MANAGER.SYS,PUB`
2. Issue a `:FILE` command for the tape drive:  
  
`FILE TAURUS;DEV=TAPE`
3. Restore the self-installing program:  
  
`RESTORE *TAURUS;TAURUSWH;SHOW`

4. Mount the Warehouse product tape and put the tape drive ONLINE. Then reply to the tape mount request.

5. After the program TAURUSWH has been restored, dismount the tape and run TAURUSWH:

```
RUN TAURUSWH
```

6. Proceed with steps 5 through 7 listed in the web download instructions above.

### **MPE/iX Installation Considerations**

The WHII program contains all Warehouse files needed to complete the installation. This program must be run from a user possessing SM (System Manager) capability.

If you are installing Warehouse on several systems, you may find it easier to DSCOPY WHII.PUB.SYS to the other systems, rather than using the tape drive. To complete the installation after moving TAURUSWH.PUB.SYS, perform the steps listed above, beginning at step 4.

If you wish to install Warehouse into an account other than TAURUS, you may run the WHII program with INFO parameter ALTACCT. (e.g. RUN TAURUSWH; INFO="ALTACCT") When run with ALTACCT, WHII prompts you for the name of account into which you wish the Warehouse files are to be installed.

If you should ever need to reinstall Warehouse, the WHII program may be run again to perform the reinstallation.

After the Warehouse installation has been completed, you may purge the WHII program since it is no longer needed.

### **Running the**

By default, the Warehouse program file is called WH and

## Warehouse client on MPE/iX

resides within the group WHII.TAURUS. To run the Warehouse client and then run a script within the client, enter:

```
:RUN WH.WHII.TAURUS
Warehouse 2.03.0001-M (c) Taurus
Software, Inc. 1997
Installed for: Taurus Software, Inc.
1> XEQ MYSCRIPT
```

If you wish to execute a particular Warehouse script at the command line, put the name of the script file in the INFO= parameter of the RUN command. For example:

```
:RUN WH.WHII.TAURUS;INFO="MYSCRIPT"
Warehouse 2.03.0001-M (c) Taurus
Software, Inc. 1997
Installed for: Taurus Software, Inc.
1> OPEN ORD IMAGE ORDB PASS=READER
MODE=5
```

If you wish, you may shorten the above by leaving off the RUN and INFO= " " as follows:

```
:WH.WHII.TAURUS MYSCRIPT
Warehouse 2.03.0001-M (c) Taurus
Software, Inc. 1997
Installed for: Taurus Software, Inc.
1. OPEN ORD IMAGE ORDB PASS=READER
MODE=5
```

## Running the Warehouse Server on MPE/iX

By default, the Warehouse Server program file is called WHSERV and resides within the group WHII.TAURUS. Be sure that the Warehouse installation has been validated before launching the Warehouse Server job. On MPE/iX, the Warehouse server is typically run as a job stream that logs on as manager.sys. It needs to be streamed by someone or something logged on as manager.sys or the job card in the job stream needs to provide the password. The job card looks like this:

```
!JOB JWHSEVR,MANAGER.SYS;HIPRI;
OUTCLASS=,1;PRI=CS
```

If your logon requires a password, add your password where pass is below:

```
!JOB JWHSEVR,MANAGER/pass.SYS;HIPRI;
    OUTCLASS=,1;PRI=CS
```

There is a sample job stream called JWHSEVR supplied with the installation

The Warehouse server program cannot be run directly after installation because it must reside in a group with PM (Privileged Mode) capability. PM capability is required by Warehouse to perform the user password checking and login that is done when making a REMOTE connection. You may solve this problem one of two ways. You may either add PM capability to the WHII.TAURUS group and account using:

```
:HELLO MANAGER.SYS
:ALTACCT TAURUS;CAP=+PM
:ALTGROUP WHII.TAURUS;CAP=+PM
```

In which case the server job could look as follows:

```
!JOB WHSERVER,MGR.TAURUS
!RUN WHSERV.WHII
!EOJ
```

Alternatively, you may copy the server program to a group that has PM capability, such as PUB.SYS. If you copy the server program, you must set the variable WHHOME to point back to the original group before running the server program. The WHHOME variable allows Warehouse to find the files necessary to run. Use SETVAR to set WHHOME to the fully qualified group name of the location of the Warehouse files and place an At sign (@) where the file name would normally go.

Example:

```
!JOB WHSERVER,MANAGER.SYS
!COPY
    WHSERV.WHII.TAURUS,WHSERV.PUB;YES
!SETVAR WHHOME, "@.WHII.TAURUS"
!RUN WHSERV.PUB
!EOJ
```

Modifying the  
authorization

Before Warehouse clients may connect to the Warehouse server, the authorization file AUTHFILE must be set up to



file on MPE/iX allow remote connections. This may be done by running either the Warehouse client program WH or the Warehouse server program WHSERV using INFO="-a". Example:

```
:RUN WH.WHII.TAURUS;INFO="-a"
```

For information on authorizing Warehouse server access and the AUTHFILE, see the section later in this chapter titled **Authorizing Warehouse Server Access**.

Stopping the Warehouse Server on MPE/iX

The Warehouse server job is stopped with the ABORTJOB command. Make sure that there is no Warehouse activity before stopping the Warehouse Server job. Example:

```
:HELLO MANAGER.SYS
:SHOWJOB JOB=@J;EXEC

JOBNUM  STATE  IPRI  ...  JOB NAME
#J55    EXEC           ...
                                WHSERVER,MGR.
                                TAURUS
JOBFENCE= 6; JLIMIT=10; SLIMIT=100

:ABORTJOB #J55
```

**Installation on  
Unix / Linux**

On Unix/Linux, Warehouse is available as a `tar` file and all files are restored using the `tar` command. All Warehouse files are installed into a new directory called `taurus/whii` within the current working directory. To install Warehouse on Unix, proceed as follows:

## Web download

1. Login to your Unix system.
2. If transferring the software distribution from another system, use binary transfer mode.
3. Transfer or copy the distribution to the directory where you wish the Warehouse files to be installed. Typical selection is a globally-accessible directory such as `/usr/local` or `/opt`.
4. If the Warehouse distribution has a `tar.gz` suffix, use `gzip` or `gunzip` to restore the tar file; if it has a `tar.Z` suffix, use `uncompress` to restore it.
5. Untar the distribution: Warehouse files will be placed in the subdirectory `taurus/whii`.
6. After the Warehouse files have been restored, the Warehouse program must be validated.
7. After the Warehouse files have been restored, the distribution tar file can be removed.

## Tape install

1. Insert the Warehouse tape into the tape drive connected to your system if installing from DAT media.
2. Using the command `cd`, change to the directory where the Warehouse files will be installed  
  
`cd /usr/local`
3. Restore the Warehouse files using the device name of your tape drive.  
  
`tar xf /dev/rmt/0m.`

4. Proceed with steps 4 through 7 listed in the web download instructions above.

**Unix/Linux  
Installation  
Considerations**

Certain browsers with certain settings will preemptively change the `tar.Z` or `tar.gz` suffix of the download package to a `tar.tar` suffix. If this happens to you, enclose the distribution package name in double quotes when performing the `Save As` operation.

When Warehouse is installed, the login user becomes the creator of all the Warehouse files.

Typically Warehouse is installed in a globally accessible directory such as `/usr/local`.

**Running the  
Warehouse Client  
on Unix/Linux**

By default, the Warehouse program file is called `warehouse` and resides within the installation directory. To run Warehouse and execute a script, enter:

```
$ /usr/local/taurus/whii/warehouse  
Warehouse 2.02.0001-H (c) Taurus  
Software, Inc. 1996  
Installed for: Taurus Software, Inc.  
1> xeq myscript
```

If you wish to execute a particular Warehouse script at the command line, put the name of the script file as a command parameter. Example:

```
$ /usr/local/taurus/whii/warehouse  
myscript
```

In order for Warehouse to execute, it must have access to the directory where the Warehouse message files are located. This is usually in the same directory as the Warehouse program and Warehouse finds the files by examining the command line that executed Warehouse. In some circumstances (such as when Warehouse is executed via a soft link, or when the Warehouse program has been moved to a

different directory than the message files) Warehouse is unable to locate the message files. If Warehouse is unable to access the message files, Warehouse prints Warehouse validation error number 20" and exits. In this event, the WHHOME environment variable may be set to direct Warehouse to the correct directory. WHHOME should be set to the directory of the Warehouse message files, with an At sign (@) placed where the message file name would go.

## Examples

### Example 1

This example creates a soft link to Warehouse, sets WHHOME using the Bourne shell, then executes Warehouse.

```
$ ln -s
  /usr/local/taurus/whii/warehouse
  wh
$ WHHOME=/usr/local/taurus/whii/@
$ export WHHOME
$ wh
Warehouse 2.03.0001-H (c) Taurus
  Software, Inc. 1997
Installed for: Taurus Software, Inc.
1> OPEN ORD ORACLE SCOTT/TIGER
```

### Example 2

This example creates a soft link to Warehouse, sets WHHOME using the C shell, then executes Warehouse.

```
% ln -s
  /usr/local/taurus/whii/warehouse
  wh
% setenv WHHOME
  /usr/local/taurus/whii/@
% wh
Warehouse 2.03.0001-H (c) Taurus
  Software, Inc. 1997
Installed for: Taurus Software, Inc.
1> OPEN ORD ORACLE SCOTT/TIGER
```

## Running the Warehouse Server on Unix

By default, the Warehouse Server program file is called whserv and resides within the installation directory. The Warehouse server must run as root

to allow it to change the client login. To enable the Warehouse server, login as `root` (or use the `su` command) and add the `s` permission to the Warehouse server program as follows:

1. Login as `root` (or use `su`)

2. Change to the Warehouse directory:

```
# cd /usr/local/taurus/whii
```

3. Change the Warehouse server owner to `root` (`uid:gid` must be `0:0`):

```
# chown root:sys whserv
```

or on Linux:

```
# chown root:root whserv
```

4. Change the `setuid` bit to run as `root`:

```
# chmod +s whserv
```

5. Before Warehouse clients may connect to the Warehouse server, the authorization file `AUTHFILE` must be set up to allow remote connections. If `AUTHFILE` does not exist, create it:

```
# touch AUTHFILE
```

6. Add `AUTHFILE` entries by running the Warehouse server program `whserv` with the parameter `-a`:

```
$ /usr/local/taurus/whii/whserv -a
```

or

```
# whserv -a
```

For more information see the section later in this chapter titled **Authorizing Warehouse Server Access**.

7. Make sure that your Warehouse installation has

been properly validated.

8. Launch the server as a daemon using `nohup`:

```
nohup /usr/local/taurus/whii/whserv &
```

The Warehouse server may be run with the `-port` `###` parameter to specify a port that is different from the default 1610. For example:

```
nohup /usr/local/Taurus/whii/whserv  
-port 20001 &
```

The Warehouse Server is now running and capable of servicing Warehouse or DataBridger Studio Client requests.

#### Stopping the Warehouse Server on Unix

The Warehouse server job is stopped with the `kill` command using the pid of the Warehouse server process. The pid can be found using the `ps -aef` command (process status). Often the `ps` command is piped to the `grep` command to search for `whserv`. The first field of the `ps` command contains the pid. Example:

```
# ps -aef | grep whserv  
root 1113      1  0   Jul 19 ?        0:00  
    /users/whii/inhouse/whserv  
# kill 1113
```

Make sure that there is no Warehouse activity before stopping the Warehouse Server daemon.

**Installation on Windows**

Taurus supports the following Windows Operating Systems:

Windows 2000/2003/2005  
Windows XP

To install Warehouse on a Windows platform, proceed as follows:

1. Logon to Windows as Administrator.
2. Download the Software from the Taurus website:

<http://www.taurus.com/support/downloads.htm>

Access to the site will require a logon ID and password that can be obtained by emailing [support@taurus.com](mailto:support@taurus.com).

The software is available in 32 and 64 bit versions and in compressed .zip format.

3. Decompress the software and run the installation program `SETUP.EXE`
4. Setup guides you through the restoration of the Warehouse files. By default, the Warehouse files are restored to location:

`C:\Program Files\Taurus\Warehouse`

or for 64 bit systems:

`C:\Program Files\Taurus\Warehouse64`

5. After the Warehouse files have been restored, the Warehouse program must be validated by running the Warehouse program `WH.EXE` with the `-v` option as follows:

Launch a Command Prompt window (MS DOS prompt) and enter:

```
C:          ← Installation drive
cd \Program Files\Taurus\Warehouse
```

```
WH -v
```

For information on validation, see the section later in this chapter titled **Validating Warehouse**.

### Windows Installation Considerations

On older Windows versions the Change Directory (`cd`) command must contain quotation marks if the path name contains a space as Program Files does in this case. Example:

```
cd "\Program Files\Taurus\Warehouse"
```

### Running the Warehouse Client on Windows

The Warehouse client program `WH.EXE` may be run from a command prompt, or it may be run from Windows by double clicking on the Warehouse icon.

If you wish to execute a particular Warehouse script from the command prompt, put the filename of the script as a command parameter. Example:

```
WH MyScript.txt
```

### Running the Warehouse Server on Windows

Before the Warehouse Server Program may be used, it must be set up as a service. Provided with the installation is a program (`instsrv`) that installs and uninstalls the Warehouse service. After the Warehouse service is installed, it must be started using the Services control panel. After the Warehouse service is started, connections may be received from Warehouse client processes.

### Installing the Warehouse Service on Windows

1. Login to your Windows system as Administrator.
2. Install the Warehouse service in one of two ways: by running the `WHSTART.EXE` program or by issuing the `instsrv` command. For convenience a `WHSTART.EXE` shortcut has been placed in the Warehouse Folder and is accessed at `START | Programs | Warehouse | Install Warehouse Service`. To issue the `instsrv`



command, launch a command prompt window and run the program with the parameters Warehouse and the fully qualified name of the WHSTART.EXE program:

```
instsrv Warehouse "c:\Program  
Files\Taurus\Warehouse\WhStart.exe"
```

3. Open the Services control panel, alphabetically locate the entry for the Warehouse service. Highlight the Warehouse service by clicking on it and click on the Start link. This starts the Warehouse service which will remain running until the system is shut down.

If you wish the Warehouse service to be started automatically at system startup, modify the Warehouse Service properties and change the Startup type to Automatic. If you do not set the service to Automatic, you will need to start the Warehouse service manually after each system restart.

4. Close related open windows. Warehouse is now ready to accept connections, but users are required to have the "Log on as a service" user right to connect to Warehouse as explained below.

#### Listening to Specific Port Numbers

On Windows systems, the Warehouse server may be directed to listen to a specific port by indicating the port number in the service name when installing the service with INSTSRV. The default Warehouse port is 1610 and is installed using something like:

```
INSTSRV Warehouse "C:\Program  
Files\Taurus\Warehouse\WhStart.exe"
```

To indicate a specific port number, append \_p#### to the service name, where #### is the port number you wish to use. For example, if you wished to use port 11610, you would install the Warehouse service as follows:

```
INSTSRV Warehouse_p11610 "C:\Program
```

Files\Taurus\Warehouse\WhStart.exe"

Port numbers are controlled by the Internet Assigned Numbers Authority. See:

<http://www.iana.org/assignments/port-numbers>

Adding the "Log on as a service" user right on Windows

Before Warehouse clients may connect to a Windows Warehouse server program, users must be given the "Log on as a service" user right.

More information can be found at:

<http://technet.microsoft.com/en-us/library/cc739424.aspx>

Windows NT

1. Run the Windows NT program User Manager for Domains. (Start | Programs | Administrative Tools (Common) | User Manager)
2. From the Policies menu, select User Rights....
3. Check the Show Advanced Rights box in the lower left.
4. Click the down arrow on the Right combo box and use the scroll bar to find and click on Log on as a service.
5. While Log on as a service is displayed as the Right, click on the Add... button.
6. A new dialog box is displayed that is used to grant the new user right to users and groups. Click on the Show users button to display users as well as groups.
7. Select a user or group that you wish to allow access with Warehouse, then click on the Add button. Repeat this process until you have added all users and groups that will access this system via the Warehouse server.

8. Click OK to return to the User Rights Policy dialog box.
9. Click OK again to return to the main program window and then exit.

**Windows 2000**

1. Run the Windows 2000 Microsoft Management Console program Local Security Policy. (Start | Programs | Administrative Tools | Local Security Policy) You may need to install the security policy snap-in. To do this run MMC.EXE from the Start | Run menu, then choose Add/Remove Snap-in... from the Console menu and install the Local Computer Policy snap-in.
2. Once the security policy editor is running expand Local Policies by clicking the + and then click on User Rights Assignment. This should give you a list of user rights.
3. Double-click on Log on as a service. This should bring up a window that shows which users have the Log on as a service right.
4. Click on Add... to the give Log on as a service to selected users and groups.
5. A new dialog box is displayed showing available users and groups.
6. Select a user or group that you wish to allow access with Warehouse, then click on the Add... button. Repeat this process until you have added all users and groups that will access this system via the Warehouse server.
7. Click OK to return to the Local Security Policy Setting dialog box.
8. Click OK again to return to the main program window and then exit by clicking on the X in the

upper right hand corner of the window.

#### Windows XP

1. Click Start | Control Panel | Administrative Tools | Local Security Policy
2. From the User Rights Assignment folder, locate the policy named Log on as a service
3. Right click on Log on as a service and select Properties
4. From the Local Security Setting tab, click on the button Add user or Group
5. From the pop up temporary window that appears, enter the user name and click the button labeled Check Names. Search until the correct user is found.
6. Click OK, and close out of all related windows. The user has now been given additional log on rights.

#### Modifying the authorization file on Windows

Before Warehouse clients may connect to the Warehouse server, the authorization file `AUTHFILE` must be set up to allow remote connections. This may done by running either the Warehouse client program `wh.exe` or the Warehouse server program `whserv.exe` with the parameter `-a`. Example:

```
WH -a
```

or

```
WHSERV -a
```

For information on authorizing Warehouse server access and the `AUTHFILE`, see the section later in this chapter titled **Authorizing Warehouse Server Access**.

**Stopping the  
Warehouse Server on  
Windows**

The Warehouse server is stopped using the Services control panel. Open the Services control panel. Alphabetically locate the Warehouse service. If the status is Started, highlight the Warehouse service by clicking on it and then click on the Stop button. This stops the Warehouse service. Press Close to exit.

**Uninstalling  
Warehouse on  
Windows**

If for any reason you wish to remove Warehouse from your system, perform the following steps:

1. Make certain that no users are accessing Warehouse.
2. Make certain the Warehouse service is stopped. See the previous section, Stopping the Warehouse Server on Windows.
3. Uninstall the Warehouse service by running the `instsrv` program with the parameter Warehouse remove:

```
instsrv Warehouse remove
```

For convenience, this has been set up as a shortcut that can be run from the Warehouse group by clicking START | Programs | Warehouse | Uninstall Warehouse Service.

4. Removal of the software can be done in one of two ways:

Access the uninstall program provided with the software: click START | Programs | Warehouse | Uninstall Warehouse

or

Open the Add/Remove Programs control panel. Alphabetically locate Warehouse and highlight the entry by clicking on it. Click on the Add/Remove button.

5. When you are asked if you want to remove it, click on **Yes**. The uninstall program is run and Warehouse is removed from your system.

**Validating  
Warehouse**

Before Warehouse can be run, it must be validated. This is done using `-v` as a parameter. (On MPE/iX, use `INFO=`). Examples:

On MPE/iX:

```
:RUN WH.WHII.TAURUS;INFO="-v"
```

On Unix:

```
$ warehouse -v
```

On Windows:

```
C: wh -v
```

After running Warehouse with the `-v` option, you are presented with a menu. If you have a demonstration copy of Warehouse and a demonstration validation code provided by Taurus Software, enter option 1. You are then asked to enter your validation code. After entering a valid code Warehouse will in a full function mode until the expiration date.

If you wish to convert a demonstration copy of Warehouse to a production copy enter option 2. You will need a production validation code that is generated by Taurus Software and is based upon the Production Validation Code displayed by Warehouse after entering a demonstration code. After entering the code to Warehouse, you are asked to enter your company name. Enter the name of you company. Note that Warehouse requires the company name to be at least 10 characters in length. If you company name is less than 10 characters long, you may add additional characters to bring it up the minimum. For example, if you company name is ACME, you may enter \*\*\*\* ACME \*\*\*\* as your company name.

**Authorizing  
Warehouse Server  
Access**

Before a Warehouse server can be run, an `AUTHFILE` must exist on the same system as the server. The `AUTHFILE` describes which client systems may connect to the server and under what conditions.

The `AUTHFILE` is edited using the Warehouse server program run with the `-a` option. See **Maintaining the `AUTHFILE`** later in this chapter.

Each `AUTHFILE` record has four fields:

Client computer:      Name or IP address of client computer. This is the computer attempting to connect to the Warehouse server.

Client user name:      User name on client computer. This is the user name of the user on the client computer that is attempting to connect to the Warehouse server.

NOTE: On MPE/iX systems, the client user name must match the `USER=` parameter of the `OPEN` statement. If passwords are required, they must be indicated in the client user name.

Examples:

`MGR.DBA/*`  
`MGR/* .DBA`  
`MGR/* .DBA/*`

Server user name:      User name on server computer. This is the user name specified in the `OPEN` or `CREATE` statement in the script running on the client



computer.

Password Req'd(Y/N): Indicates if a password is required to login. If Y, then a correct PASSWORD= must be supplied in the client OPEN or CREATE statement. If N, then no password is required when coming from the specified client computer.

Each of the first three fields can contain an asterisk (\*) used as a wildcard character to allow any match.

When a Warehouse client attempts to connect to a Warehouse server, the server searches the AUTHFILE for a record that will allow the client to login to the server. If no record is found, access is denied by the server.

#### AUTHFILE Examples

##### Example 1:

The following AUTHFILE entry is on the server RED. It allows a user logged in as MGR.GLDB on client computer BLUE.TAURUS.COM to login to the user DOUG on RED as long as the correct user password for DOUG is supplied.

Client Host Name: BLUE.TAURUS.COM  
Client User Name: MGR/\* .GLDB/\*  
Server User Name: DOUG  
Password Req'd: Y

The script on BLUE could contain the following Warehouse OPEN statement:

```
OPEN T REMOTE RED USER=DOUG PASSWORD=BLVD &  
ORACLE SCOTT/TIGER &  
HOME=/b/u01/oradata/oracle SID=ora
```

Because the "Password Req'd" field was Y, the script needed to specify DOUG's password of BLVD to RED. Note the use of "/" "\*" in the client user name.

Since, on MPE/iX, the user and password is considered as one field, the /\* is necessary in the user name to match the user name supplied in the remote connection. If no password was supplied, or the incorrect password is supplied, access is denied by the server.

#### Example 2:

The following AUTHFILE entry is on the server RED. It allows a user logged in as MGR.GLDB on client computer BLUE.TAURUS.COM to login to the user DOUG on RED, but the password does not have to be supplied.

```
Client Host Name: BLUE.TAURUS.COM
Client User Name: MGR.GLDB
Server User Name: DOUG
Password Req'd: N
```

The script on BLUE could contain the following OPEN statement:

```
OPEN T REMOTE RED USER=DOUG &
ORACLE SCOTT/TIGER &
HOME=/b/u01/oradata/ora SID=ora
```

Because the "Password Req'd" field was N, the script did not need specify DOUG's password to RED.

#### Example 3:

The following AUTHFILE entry is on the server RED. It allows any user from the domain TAURUS.COM to login to any user on RED providing the correct password is supplied.

```
Client Host Name: *.TAURUS.COM
Client User Name: *
Server User Name: *
Password Req'd: Y
```

#### Example 4:

The following AUTHFILE entry is on the server RED.

It allows any user from the Class C IP address starting with 205.178.2 to login to the server RED using the user name SALES without supplying a password.

Client Host Name: 205.178.2.\*  
Client User Name: \*  
Server User Name: SALES  
Password Req'd: N

Example 5:

The following is a list of three AUTHFILE entries:

Client Host Name: \*.TAURUS.COM  
Client User Name: \*  
Server User Name: \*  
Password Req'd: Y

Client Host Name: BLUE.TAURUS.COM  
Client User Name: MGR.GLDB  
Server User Name: DOUG  
Password Req'd: N

Client Host Name: \*  
Client User Name: \*  
Server User Name: DEMO  
Password Req'd: N

These entries indicate that anyone coming from a client in the TAURUS.COM domain may connect to the server to any user if the correct password is supplied. The second entry allows any user logged into the client computer BLUE.TAURUS.COM as MGR.GLDB to connect to the server as user DOUG without supplying a password. The final entry allows anyone coming from any system to connect as the user DEMO without a password.

Given the above AUTHFILE entries, these OPEN statements on the client would be allowed:

Logged in as MGR.GLDB on client

GREEN.TAURUS.COM:

```
OPEN T REMOTE RED USER=ANY
PASSWORD=THEPASS &
ORACLE SCOTT/TIGER &
HOME=/b/u01/oradata/ora SID=ora
```

Allowed by first AUTHFILE record.

Logged in as AUSER.SYS on client

YELLOW.TAURUS.COM:

```
OPEN T REMOTE RED USER=SWIFT PASS=EAGLE &
ORACLE SCOTT/TIGER &
HOME=/b/u01/oradata/ora SID=ora
```

Allowed by first AUTHFILE record.

Logged in as MGR.GLDB on client

BLUE.TAURUS.COM:

```
OPEN T REMOTE RED USER=DOUG &
ORACLE SCOTT/TIGER &
HOME=/b/u01/oradata/ora SID=ora
```

Allowed by second record. No password needed.

Logged in as ANYUSER on client

ROOT.HACKER.COM:

```
OPEN T REMOTE RED USER=DEMO &
ORACLE SCOTT/TIGER &
HOME=/b/u01/oradata/ora SID=ora
```

Allowed by third record. No password needed.

Given the above AUTHFILE entries, these OPEN statements on the client would *not* be allowed:

Logged in as MGR.GLDB on client

GREEN.GEMINI.COM:

```
OPEN T REMOTE RED USER=ANY PASSWORD=APASS &
ORACLE SCOTT/TIGER &
HOME=/b/u01/oradata/ora SID=ora
```

No match in AUTHFILE. Must come from  
\*.TAURUS.COM.

Logged in as MGR.GLDB on client  
ORANGE.TAURUS.COM:

```
OPEN T REMOTE RED USER=DOUG &  
ORACLE SCOTT/TIGER &  
HOME=/b/u01/oradata/ora SID=ora
```

No match in AUTHFILE. Must come from  
BLUE.TAURUS.COM.

Logged in as ANYUSER on client  
ROOT.HACKER.COM:

```
OPEN T REMOTE RED USER=DOUG &  
ORACLE SCOTT/TIGER &  
HOME=/b/u01/oradata/ora SID=ora
```

No match in AUTHFILE. Must login as user DEMO.

#### Maintaining the AUTHFILE

The Warehouse AUTHFILE is maintained by running the Warehouse server program with the "-a" option. The "-a" option causes Warehouse to be run in a special mode to edit the AUTHFILE. A menu is presented with the following options:

- 0 Writes the current authorization records to the AUTHFILE along with any changes made and exits.
1. Writes the current authorization records to the AUTHFILE along with any changes made and exits.
2. Lists the current authorization records after first reading them from the AUTHFILE.
3. Adds a new authorization record. After selecting option 2, the user is prompted for each of the four authorization fields.
4. Deletes an authorization record. After selecting option 3, the user is prompted for the record number to be deleted. Record numbers can be displayed using option 1, list.

- 99 Exits without writing changes back to the AUTHFILE.

**Checking  
Warehouse Server  
Connections**

Once a Warehouse server program is set up and running, you may wish to verify the connection from a client. This can be done by running the Warehouse client program with the `-c` (or `-connect`) option.

To check a connection to a Warehouse server, perform the following steps:

**Step 1  
Verifying Connection**

Run the Warehouse client program with the `-c` option

```
warehouse -c
```

or

```
wh -c
```

**Step 2**

Enter the name or IP address of the system (ENTER exits program):

```
Enter name or IP address of server    ->mysys
```

**Step 3**

Do not enter a user name when prompted; press ENTER instead. If a user name is entered, access validation is done. This step should be done later.

```
Enter user name on server (Optional)  -><enter>
```

**Step 4**

At this point Warehouse will verify your network connection. If you cannot connect, either the Warehouse server is not running on the remote system or you have a network problem. The error message should help diagnose the problem.

**Step 5  
Verifying Login**

If the connection test passed, re-enter the name of the system when prompted. If the connection test failed press ENTER to exit.

```
Enter name or IP address of server    ->mysys
```

**Step 6**

Enter a user that has been entered into the AUTHFILE on the remote system:

```
Enter user name on server (Optional)  ->whuser
```

- Step 7 Enter the password of the user:
- ```
Enter password ->userpw
```
- Step 8 Do not enter a database type when prompted; press ENTER instead. If a database type is entered, database access validation is done. This step should be done later.
- ```
Enter remote database type (optional) -><enter>
```
- Step 9 At this point Warehouse verifies your network connection and user access. If this step is successful you should be able to open remote connections to access remote databases and files with a Warehouse script. If this step fails, but step 4 succeeded, you may have entered something incorrectly. Check your user name and password and verify that the AUTHFILE allows access to your user from your client system. If the server is running on Windows NT, make certain the user name you entered has the "Log on as a service" user right.
- If the connection and login to the remote system was successful, a partial OPEN statement is displayed that shows an encrypted password. The encrypted password can be used to open remote connections to this system in Warehouse scripts. The password can be "cut" and "pasted" into scripts opening databases on this system. Example:
- ```
OPEN dbtag REMOTE mysys USER=whuser &  
EPASS1=f7d6ac4e5f72d2c60734ca5364272abddce &  
dbtype dbparms...
```
- Step 10 Verifying Database Access
- Once the remote system connection and login has been verified, the database connection may be verified. To verify the database connection, reenter the information entered in steps 4-7, then enter the type of the remote database. The type is case insensitive and must be either ALLBASE, ARCHIVE, CSV, FIXED, IMAGE, ODBC, ORACLE, or TEXT.
- Example:
- ```
Enter name or IP address of server ->mysys  
Enter user name on server (Optional) ->whuser
```



```
Enter password -> userpw  
Enter remote database type (optional) -> oracle
```

**Step 11**

When prompted, enter the parameters specific to the type of database. Example:

```
Enter Oracle user name -> scott  
Enter Oracle user password -> tiger  
Enter Oracle SID -> ora  
Enter Oracle HOME -> /oradata/ora
```

**Step 12**

At this point Warehouse verifies your remote database connection. If this step fails, you may have entered something incorrectly or you may not have access to the database from the login user.

If the remote database connection was successful, an OPEN statement is displayed that shows encrypted passwords and can be used to open remote connections in Warehouse scripts. This OPEN statement can be "cut" and "pasted" into scripts opening this database. Example:

```
OPEN dbtag REMOTE mysys USER=whuser &  
EPASS1=f7d6ac4e5f72d2c60734ca5364272abddce &  
oracle scott SID=ora HOME=/oradata/ora &  
  
EPASS1=7a03d4d4a62f02a95bc30a0040fe7149f792f81
```

**Environment Variables**

Warehouse installations require environment variables specific to each operating system.

**Common to all Installations**

These environment variables are common to all installations. The DB2 and ODBC libraries are dynamically loaded on UNIX platforms (except RS6000) .

COMPUTERNAME – Warehouse will use the value stored here for the Host when `gethostname()` fails.

ORACLE\_HOME – Same as Oracle HOME.

ORACLE\_SID – Same as Oracle SID.

WHCHARMAPS – the name of the CHARMAPS (extended character maps) file

WHCTLDBSFILE – the name of Warehouse control database file.

WHHOME – the location of the Warehouse home directory

WHKEEPALIVE – the value for the network SO\_KEEPALIVE used to keep a network connection active

Default: 1

WHLIBPATH – the dynamic library path

WHLOG – the name of the warehouse log file

Default: WHLOG

WHNETBUFSIZE – the size in Bytes of the network buffer

Default: 4000

WHNETTIMEOUT – enables network timeout in seconds

WHODBCHOME – the ODBC library home

WHODBCLIB – the name of the ODBC library file

Default: ODBC32.DLL

WHODBCLIBPATH – the directory where the ODBC

library files are located

WHORA7ONLY – a non-zero value indicates Oracle 7 only

WHORACLE8LIB – the OCI library name for Oracle 8

WHORACLELIB – the OCI library name for Oracle 7

WHORACLELIBPATH – the OCI library path for Oracle

WHPID – a zero value here disables PID printing

WHPROG – the Warehouse client program name

WHRBLOCKSIZE – the block size for rollback files

WHSERVERKEY – the Server Name for job scheduling

Default: `gethostname()`

## Eloquence

Eloquence (IMAGE) libraries are dynamically loaded on all platforms except MPE/iX where they are statically loaded. Variables may be set prior to running Warehouse to facilitate the dynamic loading of IMAGE (Eloquence) libraries

WHIMAGEHOME - points to the Eloquence installation directory.

Default: `/opt/eloquence6/`

WHIMAGEPATH - points to the library directory within the installation directory.

Default: `lib/pa11_32`

WHIMAGELIB - points to the library file within the directory.

Default: `libimage3k.sl`

WHSUBSYS – Override DBINFO mode 501 – subsystem access to the database

Default: `DBINFO(501)`

## HP-UX

These environment variables are common to HP-UX

installations.

UNIX95 - enables the internal `ps -p`  
WHDYNFILESYS - a value of 1 means Static file  
system, 2 means dynamic file system  
WHFILESYSLIB - the file system library name

Default: `libc.sl` or `libc.so`

WHSECLIB - the security library path  
WHTRUST - a value of 1 means this is a  
trusted system

MPE

These environment variables are common to MPE  
installations.

HPACCOUNT - the login account  
HPGROUP - the login group  
HPUSER - the login user  
WHSUBSYS - Override DBINFO mode 501 -  
subsystem access to the database

Default: `DBINFO(501)`

**Appendix A**

**Transactions and Error Handling**

<b>Transactions</b>	By default, Warehouse treats every record read and processed from an outer <code>READ</code> statement to be a single transaction. The number of records read from an outer <code>READ</code> statement that constitute a transaction may be altered by changing the <code>COMMITRATE</code> . In addition, the <code>COMMIT</code> statement may be used within the script to terminate one transaction and begin the next one.
Transaction Begin	Warehouse transactions are begun at each outer <code>READ</code> statement. The effect of a transaction begin depends upon the type of database accessed. For file types <code>ARCHIVE</code> , <code>CSV</code> , <code>FIXED</code> , <code>REPORT</code> , and <code>TEXT</code> a transaction begin has no effect. For file types <code>ALLBASE</code> , <code>DB2</code> , <code>ODBC</code> , and <code>ORACLE</code> , a transaction begin is handled by the database. For the <code>IMAGE</code> file type, database locks are placed on the dataset (or database if locking is set to <code>BASE</code> ) and if locking is set to <code>ROLLBACK</code> , <code>DBXBEGIN</code> is called. The effect of a transaction begin on a <code>REMOTE</code> database is dependent upon the type of underlying database.
Transaction End	<p>After processing each record from an outer <code>READ</code> statement (or more records if the <code>COMMITRATE</code> is set to more than 1), Warehouse commits the transaction. Warehouse also commits the transaction whenever a <code>COMMIT</code> statement is reached. To perform the commit Warehouse does a commit to each database accessed during the transaction. Databases that were not accessed during the transaction are not committed.</p> <p>The effect of a commit depends upon the type of database accessed. For file types <code>ARCHIVE</code>, <code>CSV</code>, <code>FIXED</code>, <code>REPORT</code>, and <code>TEXT</code> a commit has no effect. For file types <code>ALLBASE</code>, <code>DB2</code>, <code>ODBC</code>, and <code>ORACLE</code>, a commit does an SQL <code>COMMIT WORK</code> operation. For the <code>IMAGE</code> file type, database locks are released and if locking is set to <code>ROLLBACK</code>, <code>DBXEND</code> is called. The effect of a commit on a <code>REMOTE</code> database is dependent upon the type of underlying database.</p>
Commit Rate	The frequency of commits may be controlled by

using the `SET` statement to change the commit rate. The default commit rate of 1 causes a commit to be done after every record from an outer `READ` statement is processed. Setting the commit rate to `n` causes Warehouse to commit after every `n` records are processed from each outer `READ` statement. Larger values of `n` usually increase performance, but there is point of diminishing returns, and very large values of `n` can actually degrade system performance. In general, the commit rate should not be set above 1000.

### Transaction Rollback

In the event of a script error that is not caught by a `TRY` statement, Warehouse does a transaction rollback and then exits with a result code of 1. The *only* time an automatic rollback is done is just before Warehouse terminates due to a script error. The effect of a rollback depends upon the type of database accessed.

The effect of a rollback on a `REMOTE` database is dependent upon the type of underlying database.

ARCHIVE, CSV,  
FIXED, REPORT,  
TEXT

Rollback has no effect. (An exception is that a rollback for a `FIXED` MPE/iX message file opened with `NDR` causes the most recent record read from the message file to *remain* in the message file after Warehouse terminates.)

ALLBASE, DB2,  
ODBC, ORACLE

Rollback does an SQL `ROLLBACK WORK` operation.

IMAGE

Database locks are released and if locking is set to `ROLLBACK`, `DBXUNDO` is called.

### Script Example

The following is a sample Warehouse script that copies two tables from an `IMAGE` database to a remote Oracle database:

1. `OPEN CUST IMAGE CUSTDB.DATA &  
PASS=PW MODE=5`
2. `OPEN DEST REMOTE USYSTEM &  
USER=uuser PASS=upass &`

```
ORACLE SCOTT/TIGER &  
HOME=/u01/oradata SID=orcl  
3.  READ CM = CUST.CUST-MASTER  
4.      COPY CM TO DEST.CUSTOMERS  
5.      READ CT = CUST.CUST-TRANS &  
        FOR CUST-NO = CM.CUST-NO  
6.      COPY CT TO DEST.CUST_TRANS  
7.      ENDREAD  
8.  ENDREAD
```

Line 1: Opens the IMAGE database called  
CUSTDB.DATA.

Line 2: Opens a remote Oracle database on the  
system USYSTEM.

Line 3: Reads all records from the CUST-MASTER  
dataset. Since this an outer READ  
statement a transaction is begun for each  
record read by this statement.

Line 4: Copies the CUST-MASTER record to the  
remote Oracle table CUSTOMERS.

Line 5: Reads corresponding CUST-TRANS  
records. No transaction is begun by this  
statement.

Line 6: Copies the CUST-TRANS record to the  
remote Oracle table CUST\_TRANS.

Line 7: Terminates the CUST-TRANS read loop  
begun in line 5.

Line 8: Terminates the CUST-MASTER read loop  
begun in line 3. Since this ENDREAD  
corresponds to an outer READ statement,  
this statement causes the transaction to  
terminate which unlocks the IMAGE  
database and causes the remote Oracle  
database to do a commit.



### Error Handling

In general, there are three types of error recognized by Warehouse with each one giving different behavior. The three types of errors are:

- Script compilation errors
- Runtime expression errors
- Serious runtime errors

#### Script Compilation Errors

An error encountered during script compilation displays an error message and script compilation continues, but the script is not executed. A `GO` statement after a script compilation error causes Warehouse to exit *without* running the script.

#### Runtime Expression Errors

An error during script execution caused by bad values within an expression causes an error message to be displayed and script execution to continue. (An exception is an array bounds error which is treated as a serious error.) Expression errors are often data conversion errors and can usually be prevented by modifying your expression or data types used by the expression. When an error occurs during expression evaluation, the expression result is undefined.

If an expression error occurs within a `TRY` statement, control transfers directly to the `RECOVER` block without displaying an error message, just like a serious error.

#### Serious Runtime Errors

A serious runtime error can be caused in many ways, but is usually an error returned to Warehouse by the database. When a serious error is encountered outside a `TRY` block Warehouse performs an automatic transaction rollback, displays an error message and exits with a code of 1 (one). (On MPE/iX the job control word `CJCW` is set to 1.)

Serious errors may also be handled by specifying `ERRORS TO file` on a `COPY` statement. The `ERRORS TO file` causes any errors during the `COPY` statement to be ignored and the record that caused the error to be written to `file`.



# **Appendix B**

## **Preprocessing**

**Preprocessing**

When processing a script Warehouse preprocesses each line of the script before interpretation. All script lines are scanned for `${preprocessor-var}` and if found, the value of `preprocessor-var` is substituted. The preprocessing also handles all statements that begin with `#`. Statements beginning with `#` are statements used to manipulate preprocessing variables and determine how lines of the script are interpreted. All preprocessing is done as each line is read from the script file, that is *before* the `GO` statement.

**Preprocessing Statements**

Preprocessing statements are included in the script like any other Warehouse statement. Preprocessing statements may be continued onto the next line with an ampersand (`&`), like other Warehouse statements. The preprocessing statements are:

<code>#EXIT</code>	Exits Warehouse.
<code>#IF</code>	Conditional statement processing
<code>#PRINT</code>	Display of preprocessor variables and expressions
<code>#SETVAR</code>	Sets preprocessor variables

**#EXIT**

The `#EXIT` statement is used to stop processing and exit Warehouse. The syntax is:

```
#EXIT [exit-value]
```

Exits Warehouse returning `exit-value` to the operating system. The `exit-value` must be numeric and if omitted, zero is used. On MPE/iX, `CJCW` is set to the value of `exit-value` and if `exit-value` is unequal to zero, `JCW` is set to `FATAL`. `#EXIT` is the same as `EXIT` except that `#EXIT` allows an exit value to be passed back to the operating system.

**#IF**

The `#IF` statement is used to perform conditional

statement processing. The syntax is:

```
#IF condition [THEN]
    statements
#[ELSE [IF condition [THEN]]]
    statements
#ENDIF
```

If `condition` is `TRUE` then all script statements after `#IF` are processed. If `condition` is `FALSE` then all script statements after `#IF` are ignored until a `#ELSE` or a `#ENDIF` statement.

`#PRINT`

The `#PRINT` statement is used to display preprocessing variables and expressions. The syntax is:

```
#PRINT [exp] [, exp] [, ...]
```

`exp` is a preprocessing expression that is displayed immediately. Expressions are displayed with one space between each expression with a newline at the end. If the `#PRINT` statement ends in a comma, the newline is suppressed. A semicolon may also be used to separate expressions. If a semicolon is used as a separator, the space between expressions is suppressed.

`#SETVAR`

The `#SETVAR` statement is used to set the value of preprocessing variables. The syntax is:

```
#SETVAR varname = expression
```

The preprocessing variable `varname` is set to the value of `expression`. `varname` must not be a read-only variable such as `WHVERSION`.

## Preprocessing Expressions

Preprocessing expressions are a subset of Warehouse expressions, but are much less sensitive to data types. All operands are considered strings and are converted to integers only to perform arithmetic operations.

Identifiers

Preprocessor variables are case insensitive, must

begin with an alphabetic character or `_` and may contain any of the characters allowed in Warehouse script variables. The characters allowed are:

`A...Z, 0...9, _, +, -, *, /, ?, #, %, &, @, '.`

### Constants

Preprocessor constants may be either a string enclosed in quotes, or an integer. Preprocessor string constants work the same as Warehouse script constants, which are described in [Chapter Five, Warehouse Expressions](#). Numeric preprocessor constants must be an integer in the form of: `[- | +] digit [digit...]`

### System Variables

There are several predefined preprocessor variables that can be accessed in `#` statements. They are:

<code>FALSE</code>	False value, read-only. ( <code>\$FALSE</code> is used in a script, not by the preprocessor.)
<code>TRUE</code>	True value, read-only. ( <code>\$TRUE</code> is used in a script, not by the preprocessor.)
<code>WHERRNO</code>	Error number of most recent Warehouse error, or zero if no error has occurred. <code>WHERRNO</code> is read-write and may be set to a numeric value with <code>#SETVAR</code> . ( <code>\$ERR.WHERRNO</code> is used to access the most recent error in a script.)
<code>WHVERSION</code>	Version number of Warehouse client, read-only. The last character contains the operating system identifier. See <a href="#">Chapter One, Introduction</a> for information on Warehouse version numbers.

### Operating System Environment

Operating system environment variables may be accessed by the preprocessor by simply using the

**Variables**                      variable name as though it had been set by `#SETVAR`. If an attempt is made to access a variable not set by `#SETVAR`, the preprocessor automatically checks for an operating system variable of the same name. This allows operating system environment variables to be set outside of Warehouse and easily passed to the preprocessor. (Note that `#SETVAR` does not set operating system variables.)

**Operators**                      Preprocessor operators are as follows in order of precedence from highest to lowest:

<code>+</code> , <code>-</code>	Unary positive and negative
<code>NOT</code>	Logical not
<code>*</code> , <code>/</code> , <code>MOD</code>	Multiplication, division, and modulus
<code>+</code> , <code>-</code>	Addition and subtraction
<code>  </code> , <code>+</code>	String concatenation
<code>=</code> , <code>&lt;</code> , <code>&gt;</code> , <code>&lt;=</code> , <code>&gt;=</code> , <code>&lt;&gt;</code>	Relational operators
<code>AND</code>	Logical AND
<code>OR</code>	Logical OR

`NOT`, `AND`, and `OR` operate only on TRUE/FALSE values.

Unary `+`, `-`, `*`, `/`, and `MOD` operate only on integer values.

Relational operators (`=`, `<`, `>`, `<=`, `>=`, `<>`) and `+` operate on integer values if both operands are numeric, otherwise a string operation is done.

`||` concatenates two strings. e.g. `"12" || "34"` is `"1234"`, but `"12" + "34"` is `46`, and `"AB" + "CD"` is `"ABCD"`.

**Functions**                      Available preprocessor functions are as follows:

`ACCEPT(prompt)` Reads a string from stdin using `prompt` as a prompt string.

`DEFINED(var-name)` Returns TRUE if `var-name` is either a predefined

	preprocessor variable, a preprocessor variable set with #SETVAR, or an operating system variable set outside of Warehouse. If <code>var-name</code> is none of these <code>DEFINED</code> returns <code>FALSE</code> .
<code>DWNS(string)</code>	Downshifts all uppercase characters in <code>string</code> .
<code>LEN(string)</code>	Returns the length in characters of <code>string</code> .
<code>NOW(format)</code>	Returns the current date and time in the format specified by <code>format</code> . Date formats are described in <a href="#">Chapter Five, Warehouse Expressions</a> under the <code>DATE2STR</code> function.
<code>STR(string,index,length)</code>	Extracts substring of <code>length</code> characters from <code>string</code> starting at <code>index</code> with 1 being the first character.
<code>SYSTEM(command)</code>	Performs an operating system command <code>command</code> and returns the operating system error number. A return value of zero indicates success.
<code>UPS(string)</code>	Upshifts all lowercase characters in <code>string</code> .
<b>Preprocessor Variable Substitution</b>	Every script line is scanned for <code>\${preprocessor-variable}</code> and if any are found, the value of <code>preprocessor-variable</code> is substituted into the script line. This allows items such as passwords and file names to be calculated with the preprocessor or outside of Warehouse and substituted into the Warehouse script.



The preprocessor-variable may also be the name of an operating system environment variable.

If preprocessor-variable is has not been defined with #SETVAR and is not an operating system variable, a null string is substituted.

### Preprocessing Examples

Example 1:  
Database  
Password

The following are some examples using the Warehouse preprocessor:

```
#PRINT "IMAGE and ORACLE passwords are"
#PRINT "required to run this script."
#PRINT
#SETVAR ORAPASS = ACCEPT &
    ("Please enter Oracle password->")
#SETVAR IMGPASS = ACCEPT &
    ("Please enter Image password ->")

OPEN CUST IMAGE CUSTDB.DATA &
    PASS=${IMGPASS} MODE=5
#IF WHERRNO <> 0
# PRINT Error opening IMAGE database."
# EXIT 1
#ENDIF

OPEN DEST ORACLE SCOTT/${ORAPASS} &
    HOME=/u01/oradata SID=orcl
#IF WHERRNO <> 0
# PRINT Error opening Oracle database."
# EXIT 2
#ENDIF

READ CM = CUST.CUST-MASTER
    COPY CM TO DEST.CUSTOMERS
    READ CT = CUST.CUST-TRANS &
        FOR CUST-NO = CM.CUST-NO
            COPY CT TO DEST.CUST_TRANS
        ENDREAD
    ENDREAD
```

The above script prompts the user for the Image and Oracle database passwords and uses the passwords entered to open the databases. Success of the OPEN statements is checked using WHERRNO and if either database open fails, processing is stopped using #EXIT.

Example 2:  
Creating a File by  
Date

```
#SETVAR FNAME = "F" || NOW("YYMMDD")
CREATE LOGFIL FIXED ${FNAME}

OPEN ODB ORACLE SCOTT/TIGER &
    HOME=/u01/oradata SID=orcl

READ CL = ODB.CUST_LOG
    COPY CL TO LOGFIL
ENDREAD
```

This script calculates a file name in the form of Fyymmdd to which customer log records are copied. The file is then opened with the CREATE statement and customer log records are copied from the Oracle database to the file.

Example 3:  
Opening a Remote  
or Local Database

```
#IF DEFINED(COMPUTERNAME)
#  IF UPS(COMPUTERNAME) = "SYSTEM04"
#    SETVAR LOCAL = TRUE
#  ELSE
#    SETVAR LOCAL = FALSE
#  ENDIF
#ELSE
#  SETVAR LOCAL = FALSE
#ENDIF

#IF LOCAL
OPEN ODB ORACLE SCOTT/TIGER &
    HOME=/u01/oradata SID=orcl
#ELSE
OPEN ODB REMOTE SYSTEM04 &
    USER=WHUSER PASS=WHUPASS &
    ORACLE SCOTT/TIGER &
    HOME=/u01/oradata SID=orcl
#ENDIF

READ CM = CUST.CUST-MASTER
    COPY CM TO DEST.CUSTOMERS
    READ CT = CUST.CUST-TRANS &
    FOR CUST-NO = CM.CUST-NO
        COPY CT TO DEST.CUST_TRANS
    ENDREAD
ENDREAD
```

This script determines if it is being run on SYSTEM04 by looking at the operating system environment variable COMPUTERNAME, after first checking that it is defined. If it being run on

SYSTEM04 a local Oracle database is opened,  
otherwise a remote Oracle database is opened on  
SYSTEM04.



**Appendix C**  
**Character Maps**

**Character Map Files**

To support the `CMAP` function, Warehouse relies on underlying "charmap" files that define each character set. The charmap files are in an industry standard format and are commonly available on Unix platforms and on the internet (For example, charmaps are available at: <http://std.dkuug.dk/i18n/charmaps.646/>). (On a Unix system, see "man charmap" for details.) The charmap files used by Warehouse must be listed in the `CHARMAPS` file. The `CHARMAPS` file is created by the user in a text editor and lists the names of the charmap files Warehouse is to access, one file name per line. The `CHARMAPS` file must reside in the same directory as the Warehouse program.

For example, if your Warehouse program is called

```
/usr/local/taurus/whii/warehouse
```

then your `CHARMAPS` file must be called:

```
/usr/local/taurus/whii/CHARMAPS
```

**Sample Warehouse `CHARMAPS` File**

Your `CHARMAPS` file might contain the following two lines pointing to your two charmap files:

```
# Warehouse CHARMAPS file
/usr/lib/localedef/src/iso_8859_1/charmap.src
/usr/local/charmaps/DIN_66003
```

**Charmap File Header**

Each charmap file has a header and a body. The header has the following directives:

```
<code_set_name> CHARSET-NAME
```

`CHARSET-NAME` is the name of the character defined. This is the name used by the `CMAP` function. It may be enclosed in quotation marks. This directive is required.

```
<comment_char> COMMENT-CHARACTER
```

`COMMENT-CHARACTER` is the character used to

indicate a comment line with the character map file. This directive is optional and the default comment character is a pound sign (#).

<escape\_char>      ESCAPE-CHARACTER

ESCAPE-CHARACTER is the character used to indicate an escape where the following character is interpreted as part of a token instead of as a special character. The escape character is also used to indicate a numeric character value. This directive is optional and the default escape character is a backslash (\).

<mb\_cur\_max>      MAXIMUM-BYTES-PER-CHARACTER

MAXIMUM-BYTES-PER-CHARACTER indicates the maximum number of bytes per character. This value must be 1, 2, or 3. This directive is required.

<mb\_cur\_min>      MINIMUM-BYTES-PER-CHARACTER

MINIMUM-BYTES-PER-CHARACTER indicates the minimum number of bytes per character. This value must be 1, 2, or 3 and not greater than the maximum bytes per character. This directive is required.

### CHARMAP File Body

The body of the character map immediately follows the header and is as follows:

```
CHARMAP
<id1>  value1  optional-comment-1
<id2>  value2  optional-comment-2
<id3>  value3  optional-comment-3
...
CHARMAP  END
```

The beginning of the body must begin with CHARMAP on a line by itself and must end with CHARMAP END on a line by itself. (END CHARMAP is also recognized.) In between are the definitions for each character in the character map.

`idN` is the id for each character. The id must be enclosed in angle brackets (<>). When translating, the id of each source character is matched with the id of target character to produce the target value. A range of ids may be specified using the syntax: `<idNN>...<idNN> valueN`. For example:  
`..<ct10>...<ct131> \x00  
 defines 32 sequential ids: <ct10>, <ct11>, ..., up to <ct131>.`

`valueN` is the value for each character. The value may be indicated in decimal using `\dnn`, octal using `\nnn`, or hexadecimal using `\xnn`. For example, an uppercase A could be indicated using `"\d65"`, `"\101"`, or `"\x41"`. Two byte values are specified in decimal using `\dnn\dnn`, in octal using `\nnn\nnn`, or hexadecimal using `\xnn\xnn`. A given value may be specified by more than one id.

`optional-comment-N` is an optional description of the character.

**Sample CHARMAP File** Here is a complete sample charmap file:

```
#
# POSIX Charmap source file
#
<code_set_name>      posix
<mb_cur_max>         1
<mb_cur_min>         1

CHARMAP

<NUL>                \x00
<SOH>                \x01
<STX>                \x02
<ETX>                \x03
<EOT>                \x04
<ENQ>                \x05
<ACK>                \x06
<alert>              \x07
<backspace>          \x08
<tab>                \x09
<newline>            \x0a
<vertical-tab>        \x0b
<form-feed>          \x0c
```



<carriage-return>	\x0d
<SO>	\x0e
<SI>	\x0f
<DLE>	\x10
<DC1>	\x11
<DC2>	\x12
<DC3>	\x13
<DC4>	\x14
<NAK>	\x15
<SYN>	\x16
<ETB>	\x17
<CAN>	\x18
<EM>	\x19
<SUB>	\x1a
<ESC>	\x1b
<IS4>	\x1c
<IS3>	\x1d
<IS2>	\x1e
<IS1>	\x1f
<SP>	\x20
<space>	\x20
<exclamation-mark>	\x21
<quotation-mark>	\x22
<number-sign>	\x23
<dollar-sign>	\x24
<percent-sign>	\x25
<ampersand>	\x26
<apostrophe>	\x27
<left-parenthesis>	\x28
<right-parenthesis>	\x29
<asterisk>	\x2a
<plus-sign>	\x2b
<comma>	\x2c
<hyphen>	\x2d
<hyphen-minus>	\x2d
<period>	\x2e
<full-stop>	\x2e
<slash>	\x2f
<solidus>	\x2f
<zero>	\x30
<one>	\x31
<two>	\x32
<three>	\x33
<four>	\x34
<five>	\x35
<six>	\x36
<seven>	\x37
<eight>	\x38
<nine>	\x39
<colon>	\x3a

<semicolon>	\x3b
<less-than-sign>	\x3c
<equals-sign>	\x3d
<greater-than-sign>	\x3e
<question-mark>	\x3f
<commercial-at>	\x40
<A>	\x41
<B>	\x42
<C>	\x43
<D>	\x44
<E>	\x45
<F>	\x46
<G>	\x47
<H>	\x48
<I>	\x49
<J>	\x4a
<K>	\x4b
<L>	\x4c
<M>	\x4d
<N>	\x4e
<O>	\x4f
<P>	\x50
<Q>	\x51
<R>	\x52
<S>	\x53
<T>	\x54
<U>	\x55
<V>	\x56
<W>	\x57
<X>	\x58
<Y>	\x59
<Z>	\x5a
<left-square-bracket>	\x5b
<backslash>	\x5c
<reverse-solidus>	\x5c
<right-square-bracket>	\x5d
<circumflex>	\x5e
<circumflex-accent>	\x5e
<underscore>	\x5f
<underline>	\x5f
<low-line>	\x5f
<grave-accent>	\x60
<a>	\x61
<b>	\x62
<c>	\x63
<d>	\x64
<e>	\x65
<f>	\x66
<g>	\x67
<h>	\x68

<i>	\x69
<j>	\x6a
<k>	\x6b
<l>	\x6c
<m>	\x6d
<n>	\x6e
<o>	\x6f
<p>	\x70
<q>	\x71
<r>	\x72
<s>	\x73
<t>	\x74
<u>	\x75
<v>	\x76
<w>	\x77
<x>	\x78
<y>	\x79
<z>	\x7a
<left-brace>	\x7b
<left-curly-bracket>	\x7b
<vertical-line>	\x7c
<right-brace>	\x7d
<right-curly-bracket>	\x7d
<tilde>	\x7e
<DEL>	\x7f
<HIBIT128>...<HIBIT255>	\x80
END CHARMAP	

- , 225, 433
- ! statement, **105**, 265
- # Statements, 430
- \$CENTER, 55
- \$DATASETS, **155**
- \$ERR, 219
- \$ERR System Variable**, 220
- \$ERR.ESCMMSG, 42
- \$FALSE, 219
- \$FALSE., 307, 361
- \$HOUR, 219
- \$MYPID, 219
- \$NEW, 55
- \$NOW, 70, 219
- \$NOW0, 219
- \$NULL, 219
- \$PAGE, 67
- \$PAGENO, 55
- \$RECNUM, 219
- \$TAB, 55, 67
- \$TODAY, 219
- \$TRUE, 219, 307, 361
- \$UNKNOWN, 219, 234
- (BACKWARDS), 152
- (RECNUM), 152
- \*, 224, 225, 433
- \* statement, **106**
- .ini file, 390
- /, 225, 433
- //, 15
- { }, 14
- || (string concatenation), 225, 433
- +, 225, 433
- <, 225, 433
- <=, 225, 433
- <>, 225, 433
- =, 225, 433
- ==, 225, 234
- >, 225, 433
- >=, 225, 433
- ABORTJOB, 395
- ABS, **236**
- ACCEPT, **236**, 433
- Addition, 225
- ALLBASE, 20, 63, 80, **109**, 190
- ALLBASE BINARY data type, **275**
- ALLBASE CHAR data type, **276**
- Allbase data types, 275
- ALLBASE DECIMAL data type, **277**
- ALLBASE DOUBLE PRECISION data type, **279**
- ALLBASE FLOAT data type, **281**
- ALLBASE INTEGER data type, **282**
- ALLBASE REAL data type, **282**
- ALLBASE SMALLINT data type, **284**
- ALLBASE VARBINARY data type, **285**
- ALLBASE VARCHAR data type, **286**
- ALLOW NULLS, 275, 304, 326, 342
- AND, 225, 433
- ARCHIVE, 20, 30, 63, 80, **113**, 188, 190
- archive file, 20, 30, 63, 80
- array, 93
- ARRAY, **359**
- Array comparison, 227
- ARRAY data type, **359**
- Array identifiers, 216
- array index, 226
- ARRAYIFY, 85, **86**, 90
- ASC, 71, 72
- ASCII, 237
- authfile, 382
- AUTHFILE, 183, 186, 189, 381, **410**
- AUTOCOMMIT, 169
- AUTOPAD, 82, 230, 256
- backwards read, 153
- BAND, 225
- BINARY, **361**
- BINARY data type, **361**
- Bit Not, 224
- BLOB, 380, 389
- BOOLEAN, **236**, 247, **361**
- BOOLEAN data type, **361**
- BOR, 225
- BSL, 225
- BSR, 225
- Built-In Functions**, 236
- BULK, 181
- BXOR, 225
- c, 375
- CALL**, 16, 47, 48, 242
- capture file, 143
- carriage control, 197
- centering output, 55
- CHAR, **362**
- CHAR data type, **362**
- charmap file, 237
- CHARMAPS file, 440
- CHARSET, 32, 33, 36, 85, **86**, 91, 229, 440
- CHONOS, **362**
- CHONOS data type, **362**
- CHR, **237**, 256
- CIU, 157
- client, 375

- CLOB, 380, 389
- CLOSE, 18**
- CMAP, **237**
- CMAP function, 440
- COBOL PICTURE, 67
- comma separated value files, 119
- comment, 106**
- comments, 15
- COMMIT, 20, 80, 83, 424**
- COMMITRATE, 82, **83, 424**
- connect, 375
- Constants, 218**
- CONVERT, 227, 230, **238, 243**
- COPY, 23, 44, 109, 113, 119, 131, 137, 147, 163, 176, 183, 199**
- CREATE, 30, 109, 114, 119, 131, 137, 148, 165, 178, 186, 196, 199**
- critical item update, 157
- CSV, 20, 23, 30, 63, 80, **119, 188, 190**
- Current date, 219
- Customer Support, 6
- Data Encryption Standard, 133, 151, 166, 178, 187, 190
- data types, 274
- database tag, 18, 30, 60, 63, 85, 94, 110, 113, 114, 115, 132, 150, 166, 178, **215**
- DataBridger Studio, 8, 133, 151, 166, 178, 187, 190
- datagram, 270, 271
- datasets, 155
- DATE, 363**
- Date Addition, 232
- date calculations, 241
- Date Comparison, 233
- DATE data type, **363**
- Date Operators, 232**
- Date Subtraction, 232
- DATE2STR, **239, 261**
- DATETIME, **364**
- DATETIME data type, **364**
- DAYNUM, **241, 272**
- DB2, 63, **131**
- DB2 data types, 342
- DB2DIR, 133
- DB2INSTANCE, 133
- DBUTIL.PUB.SYS, 157
- DBXBEGIN, 158, 424
- DBXEND, 20, 80, 158, 424
- DBXUNDO, 158, 425
- DEC, 277, 309, 345
- default Warehouse port, 403
- DEFER, 85, **86, 157**
- DEFINE, 32, 44, 49**
- DEFINED, 433
- DELETE, 38, 109, 115, 124, 132, 140, 148, 165, 178, 188, 196, 202**
- demonstration validation, 409
- DESC, 71, 72
- DIRECT, 39, 242**
- DIVF, 225
- DIVI, 225
- division, 226
- Division, 225
- double byte character strings, 251, 256
- DWNS, **242, 272, 433**
- EBCDIC, 238
- ELSE, **58**
- email address, 6
- encrypted password, 133, 151, 166, 178, 187, 190, 418, 419
- END, 41**
- ENDFUNCTION, 41
- ENDIF, 41, 58
- ENDREAD, 11, 41, 71
- ENDTRY, 41
- ENDWHILE, 41, 102
- environment variable, 245
- environment variables, 432
- Equal, 225
- error handling, 99
- ERRORS TO, 25, 184
- ESCAPE, 42, 99, 243**
- Examples, 234
- EXIT, 43, 430**
- expressions, 214
- EZ-Install/3000, 391
- FALSE, 432
- FAQ, 6
- FAX, 6
- FIELD, **243**
- file tag*, 30, 54, 66, 120, 124, 137, 140, 196, 199, 202
- FILL, **244**
- FILLR, **245**
- FIXED, 20, 23, 30, 63, 80, **137, 188, 190**
- fixed length record, 137
- FLOAT, **364**
- FLOAT data type, **364**
- floating point divide, 226
- FOR, 11, **71, 110, 116, 128, 133, 144, 152, 167, 179, 192, 204**
- FORMAT, 23, 32, **44, 48, 71, 97, 103, 114, 116, 128, 144, 152, 183, 192, 204**
- FUNCTION, 34, **47**
- GETENV, **245**
- global variable, 34

- GO**, 53, 427
- Greater than, 225
- Greater than or equal, 225
- GUI, 8
- handles, 169
- HASH, 246
- HEADER**, 54, 196
- HELP**, 57
- HP3000 floating point, 298, 299
- Identifiers**, 215
- IEEE floating point, 279, 281, 282, 288, 290, 310, 317, 347, 351
- IF**, 58, 246, 430
- IMAGE, 20, 63, 80, 147, 190
- IMAGE data types, 288
- IMAGE E2 data type, 288
- IMAGE E4 data type, 290
- IMAGE I1 data type, 291
- IMAGE I2 data type, 292
- IMAGE In data type, 293
- IMAGE J1 data type, 291
- IMAGE J2 data type, 292
- IMAGE Jn data type, 293
- IMAGE K1 data type, 295
- IMAGE K2 data type, 296
- IMAGE Kn data type, 293
- IMAGE P data type, 297
- IMAGE R2 data type, 298
- IMAGE R4 data type, 299
- IMAGE U data type, 300
- IMAGE X data type, 301
- IMAGE Z data type, 302
- IMAGE\_, 150
- Implicit Type Conversion, 230
- ini file, 390
- Installation on MPE/iX**, 391
- Installation on Unix/Linux**, 396
- Installation on Windows**, 401
- instsrv, 402
- INTEGER**, 365
- INTEGER data type, 365
- integer divide, 226
- INTERVAL**, 366
- INTERVAL data type, 366
- IP address, 186, 189
- ISBOOLEAN, 247
- ISDATE, 247, 264
- ISDIGITS, 248
- ISNUMERIC, 249
- ISNUMP, 250
- ISNUMZ, 251
- kill, 400
- LEN, 251, 433
- Less than, 225
- Less than or equal, 225
- line continuation, 14
- LIST**, 60, 95
- local variable, 34
- LOCK, 148, 158
- LOCKING, 157
- locking hints, 164
- Log on as a service, 404
- Logon as a service
  - Windows 2000, 405
  - Windows NT, 404
  - Windows XP, 406
- LONGSIZE, 135, 169, 181
- MAGICON, 227, 252
- MATCH**, 253
- MAXHANDLES, 169
- MAXOPENS, 159
- message files, 143
- MOD, 225, 433
- Modulus, 225
- MPE/iX Files, 142
- MSG, 143
- MSGs, 82, 83
- Multiplication, 225
- native character, 313, 315, 316, 366
- NDR, 20, 80, 143
- negative, 224
- non-destructive read*, 143
- nostats, 85, 375
- not, 224
- NOT, 224, 433
- Not equal, 225
- NOW, 433
- NOWAIT, 23, 184
- NSTRING, 366
- Null Comparisons, 234
- Null Operations**, 234
- NUMERIC, 238, 249, 254, 265, 277, 309, 345, 367
- numeric constants, 218
- NUMERIC data type, 367
- NUMP**, 255
- NUMZ**, 256
- ODBC, 20, 63, 80, 163, 190
- ODBC BIGINT data type, 305
- ODBC BINARY data type, 305
- ODBC BIT data type, 306
- ODBC CHAR data type, 307
- ODBC data types, 304
- ODBC DATE data type, 308
- ODBC DECIMAL data type, 309

- ODBC DOUBLE PRECISION data type, **310**
- ODBC driver, 163
- ODBC INTEGER data type, **312**
- ODBC LONG NVARCHAR data type, **313**
- ODBC LONG VARBINARY data type, **312**
- ODBC LONG VARCHAR data type, **314**
- ODBC NCHAR data type, **315**
- ODBC NVARCHAR data type, **316**
- ODBC REAL data type, **317**
- ODBC SMALLINT data type, **319**
- ODBC TIME data type, **319**
- ODBC TIMESTAMP data type, **320**
- ODBC TINYINT data type, **322**
- ODBC trace, 169
- ODBC UNIQUEIDENTIFIER data type, **323**
- ODBC VARBINARY data type, **323**
- ODBC VARCHAR data type, **324**
- ODBC32 control panel, 163
- ODBCTRACE, 169
- OFFSET, 368
- Omnidex, 152
- OPEN, 8, **63**, 110, 115, 124, 132, 140, 150, 166, 178, 189, 197, 202, 418, 419
- Operations on nulls, 234
- Operators**, 224
- OR, 225, 433
- Oracle, 63
- ORACLE, 20, 63, 80, **176**, 190
- ORACLE CHAR data type, **326**
- Oracle data types, 326
- ORACLE DATE data type, **327**
- ORACLE FLOAT data type, **328**
- ORACLE INTERVAL, **329**
- ORACLE LONG data type, **331**
- ORACLE LONG RAW data type, **333**
- ORACLE LONG RAW\_, 333
- ORACLE LONG\_, 331
- ORACLE NUMBER data type, **335**
- ORACLE RAW data type, **337**
- Oracle sequence, 176
- ORACLE TIMESTAMP data type, **338**
- ORACLE VARCHAR2 data type, **340**
- ORACLE VARCHAR2\_, 340
- ORACLE\_HOME, 179
- ORACLE\_SID, 179
- ORD, 237, **256**
- ORDER BY, **71**, 110, 116, 128, 133, 144, 152, 167, 179, 192, 204
- Order of Precedence, 228
- packed decimal, 297
- PAD, 82, 230, **256**
- PAGE, 66
- page header, 54
- page number, 55
- PAGELength, 82, **84**, 198
- PAGEWIDTH, 82, **84**, 198
- password, 187, 190
- pause, 375
- PIC, 54, 66, **67**, 225
- picture, 67
- port, 186, 189
- port 1610, 382
- port n, 382
- POS, **257**
- positive, 224
- Preprocessing Statements, 430
- PRINT, 55, **66**, 431
- Print Picture, 225
- PRINTNULL, 82, **84**, 91
- Process ID, 381
- Product Version**, 5
- production validation, 409
- PROGRESS, **84**
- PROGRESS, 82
- ps, 400
- READ, 11, 38, 44, **71**, 101, 110, 115, 128, 133, 144, 152, 167, 179, 192, 197, 204
- read loop, 8, 11
- read tag, 11, 23, 38, 101, **216**
- read-tag, 71
- RECNUM, 159
- RECNUMS, 85, **87**
- RECORD, **367**
- Record comparison, 227
- Record identifiers, 216
- RECORD type, **367**
- RECOVER, 42, **99**
- Remainder, 225
- REMOTE, 20, 30, 63, 80, **183**, 381
- REPLACE, **257**
- REPORT, 20, 63, 80, **196**
- RETURN, 48, **78**
- ROLLBACK**, 80, 158, 424, 425
- ROUND, **259**
- Schema name, 133, 166
- SCRUB, **259**
- Self Service Portal, 6
- sequence, 176, 177
- server, 381
- serverinfo, 383
- Services control panel, 403, 407
- SET**, 82, 118, 130, 134, 145, 156, 168, 180, 194, 197, 205, 390
- SETVAR, 32, **92**, 230, 431
- SHOW, 60, **94**, 170

- showinfo, 376, 383
- SHOWSQL, 170, 181
- showversion, 375, 383
- SIGNED, **369**
- SIGNED data type, **369**
- SIZEOF, **259, 270**
- SLEEP, **260**
- sort order, 72
- SQL, 275
- SQL BINARY data type, **342**
- SQL CHAR data type, **343**
- SQL data types, 131, 342
- SQL DATE data type, **344**
- SQL DATE\_, 344
- SQL DECIMAL data type, **345**
- SQL DOUBLE PRECISION data type, **347**
- SQL DOUBLE PRECISION\_, 347
- SQL INTEGER data type, **348**
- SQL INTEGER\_, 348
- SQL LONG VARBINARY data type, **349**
- SQL LONG VARCHAR data type, **350**
- SQL REAL data type, **351**
- SQL REAL\_, 351
- SQL Server, 63, 163
- SQL Server **uniqueidentifier**, 323
- SQL SMALLINT data type, **352**
- SQL SMALLINT\_, 352
- SQL statements, 39, 242
- SQL TIME data type, **353**
- SQL TIME\_, 353
- SQL TIMESTAMP data type, **354**
- SQL TIMESTAMP\_, 354
- SQL VARBINARY data type, **356**
- SQL VARCHAR data type, **357**
- start, 85, 97, 375
- START, 82, **85, 97**, 103, 375
- STATS, 82, **85**
- stdin, 236
- STR, **260**, 433
- STR2DATE, 239, 247, **261**
- STRING, 238, 254, **265, 370**
- String concatenation, 225
- String constants, 218
- String Constants, 218
- STRING data type, **366, 370**
- Subtraction, 225
- Superdex, 152
- SYSTEM, 105, **265**, 433
- system command**, 105
- System Constants**, 219
- TAB, 54, 66
- tab position, 55, 67
- tar, 396
- TCP/IP, 183
- TEXT, 20, 23, 30, 63, 80, 188, 190, **199**
- text files, 199, 206
- third party indexing, 152
- Third Party Indexing, **154**
- TIME, **370**
- TIME data type, **370**
- time zone, 220
- tns, 65
- TOKEN, **266**
- TOKENCOUNT, **267**
- TPI, 152, 159, See Third Party Indexing
- TRANS, 170
- transaction, 83
- Transaction Files, 143
- TRIML, 251, **268**
- TRIMR, 251, 268
- TRUE, 432
- TRUNC, **268**
- TRY, 24, **99**, 184, 242, 243, **269**, 425, 427
- type family, 230
- UDP, 270, 271
- UDPRECV, **270**
- UDPSEND, **271**
- unary negative, 224
- unary positive, 224
- UNLOCK, 158, 160
- UNSIGNED, **371**
- UNSIGNED data type, **371**
- UPDATE**, **101**, 111, 118, 130, 135, 146, 161, 174, 182, 194, 198, 205
- UPS, 242, **272**, 433
- user right, 404
- user-defined function, 16, **47**, 78
- v, 376
- v option, 409
- validate, 376
- validation error number 20, 397
- variable, 215
- variable record length, 142
- Variable substitution, 434
- version number, 5
- WAIT, 24, 184
- Warehouse 1 archive file, 115
- Warehouse data types, 359
- Warehouse server, 183
- Warehouse Server, 393, 398
- Warehouse Service Installation on Windows, 402
- WHERRNO, 432
- WHHOME, 394, 397
- WHILE**, **102**



## Index

---

WHPROCS, 381  
WHSPID, 381  
WHSTART.EXE, 402  
WHVERSION, 432  
writing records, 23  
XEQ, 85, 97, **103**, 375  
YYYYMMDD, 241, **272**  
zoned decimal, 302

