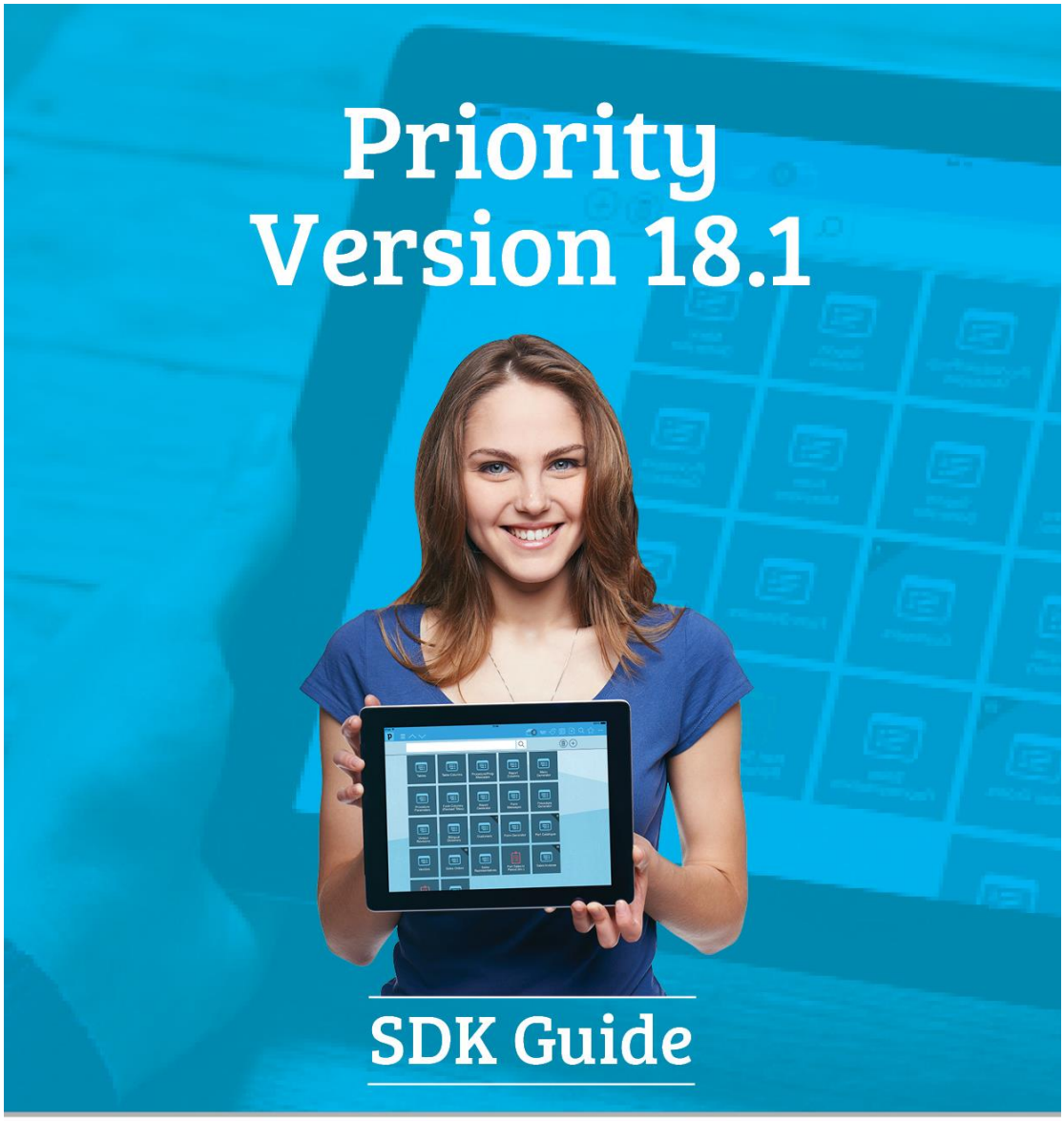

priority™

The information contained in this manual is subject to change without notice. We assume no responsibility for errors or omissions, nor is liability assumed for damages resulting from the use of the material contained herein.

Priority™ is a trademark of Priority Software Ltd.

http://www.priority-software.com/

Contents

Preface

This manual is meant to teach the programmer to create customizations of ***Priority*** using our Software Development Kit (SDK).

Before you begin, it is essential to be completely familiar with the user interface (see the ***Priority*** **User Interface Guide**, available in the Customer Zone of the ***Priority*** web site). It is also important to be well versed in basic SQL concepts and commands.

Chapter 1:

Constructing and Modifying Database Tables

Tables

Table Names

The following rules apply to table names:

- They are restricted to 20 characters.
- They may only contain alphanumeric values and the underline sign (no spaces).
- They must begin with a letter.
- You may not use a reserved word (a list of reserved words appears in the *Reserved Words* form — *System Management* → *Dictionaries*).
- The name of any new table (one you have added yourself) must begin with a four-letter prefix (e.g., **XXXX_CUSTOMERS**). All tables (and any other ***Priority*** entities) that you have created for the same customer should share the same prefix.

Table Type

The table type determines whether the table is an application table or a system table. An application table is a table in which data is maintained separately for each ***Priority*** company, whereas a system table is used to store data that is common to all companies in the current ***Priority*** installation.

For historical reasons, ***Priority*** lists 4 possible values in the **TYPE** column:

- 0/1 – for application tables
- 2/3 – for system tables

However, new tables should always be assigned type 1.

Note: You cannot create new system tables or add new columns to system tables.

Columns

Column Names and Titles

The column name is unique to its table, and is used in SQL statements. The column title is also unique to its table; it is updatable and is utilized in the user interface (forms, reports, programs, ODBC).

Every column must be assigned both a name and a title. For instance, the column which stores an order number has the name **ORDNAME** and the title *Order Number*.

A title may be easily changed (even translated into another language) as often as necessary. In contrast, a change in column name will require appropriate

changes in SQL statements that refer to the column (e.g., in form triggers, compiled programs). As the column name is used in SQL statements, it is subject to the same restrictions as table names (see below).

Column Types

The following table lists all available column types:

Col. Type	Description	Width	Form Col. Type
	string of characters	>1	String→
	single character	=1	Character
REAL	real number	any	Real
INT	signed integer	any	Integer
	date (mm/dd/yy or dd/mm/yy)	8	Date
	date (mm/dd/yyyy or dd/mm/yyyy)	10	Date
	date & time (24-hour clock)	14	Date+Time
	time (24-hour clock)	5	hh:mm
	span (number of hours and minutes)	6	hhh:mm
DAY	day of the week	3	Day

Note: It is important to distinguish between integers (columns of **INT** type, e.g., **QUANT**, **BALANCE**) and strings of digits (columns of **CHAR** type, e.g., **ZIPCODE**, **ORDNAME**, **PHONE**).

With one exception (see below), you cannot change the type of an existing column. Instead, you need to take the following steps:

1. Add a new column of the correct type.
2. Write a short command to filter data from the existing column into the new one.
3. Delete the old column.

Exception to the above rule: During the development phase, you can convert **INT** columns to **REAL** and vice versa, using the *Change Number Type* program (see p. 8). However, once a custom development has been installed in your working environment, this operation may fail, in which case you should use the above method instead.

Decimal Precision

Decimal precision (the number of places to display to the right of the decimal point) is optional; it is used in real numbers and shifted integers.

A shifted integer is stored in the database as an integer but is displayed as a real number (see, for example, the **TQUANT** column in the **ORDERITEMS** table).

Note: When working with shifted integers, use the REALQUANT function to retrieve the actual value.

Rules for Columns

- Column names are up to 20 characters.
- Column names must be made up of alphanumeric values and the underline sign (no spaces).
- Column names must begin with a letter.
- The column name may not be a reserved word (a list of reserved words appears in the **RESERVED** form).
- When adding a new column to a standard table, you must assign the column name a four-letter prefix (e.g., **XXXX_CUSTNAME**). This should be the same prefix you use for all entities that you add to ***Priority*** for the customer in question.
- Column titles (up to 20 characters, including spaces) must be enclosed in single quotations, e.g., '*Order Number*'.
- Decimal precision can only be specified for a **REAL** or **INT** column. Most columns have a decimal precision of 2. To designate a **REAL** number with indefinite precision, use decimal precision 0; otherwise, the number will be rounded up to a defined precision by INSERT and UPDATE statements. In the case of a shifted integer, decimal precision must be equal to the value of the DECIMAL system constant (or it may be 0, i.e., a regular integer).
- Modification of a column affects all forms and reports in which the column appears.
- You cannot delete a column that appears in a form, report or procedure.
- You can only change the following column types: **INT** to **REAL** and vice versa (number conversion), and only during the development phase.
- Text columns (i.e., **CHAR** columns) should not exceed a width of 80 characters. Wider columns might not be displayed well in forms, depending on the screen resolution of the user's computer. While there are a few table columns whose width exceeds 80 characters (e.g., the **MESSAGE** column in the **ERRMSGS** table), these columns are generally only displayed in reports. If you need a wide text column, it is recommended that you use a sub-level text form instead.
- You cannot add columns to system tables.

Keys

The last step in table construction is to specify the keys attached to the table, the columns that make up each key, and the priority of each column within the key. Keys are also assigned priorities, but this is only relevant for certain types (see below). The order in which keys and key columns are designated determines their priorities.

Keys are used to provide access to records in the table. Autounique and unique keys ensure that no two records in a given table will have identical values in the columns making up these keys.

The Autounique Key

The autounique key is similar to the identity column in MSSQL or sequence in Oracle. It allows you to create a column that will automatically distinguish

between all records in the table. This column automatically receives a unique integer each time a record is added.

Example: The autounique key comprised of the **CUST** column assigns a unique internal number to each customer. This is distinguished from the user-assigned customer number (**CUSTNAME**) that appears in the forms.

The autounique key is an effective means of joining tables in a form. For example, the customer number appears both in the *Customers* form and the *Orders* form. Rather than joining the **CUSTOMERS** and **ORDERS** tables through the customer number, they are joined through the **CUST** column. The advantage is twofold: First, the unique value of the **CUST** column, which joins the tables, always remains constant, while the customer number, which is only stored in a single table (**CUSTOMERS**), may be changed at will. Thus, if at some stage, a decision is made to add a “P” to the numbers of preferred customers, the revision will only be made in the records of the **CUSTOMERS** table. The **ORDERS** table will not be affected at all.

Similarly, a decision to modify the width of the **CUSTNAME** column would only require a change in the **CUSTOMERS** table. Had the join been made through the **CUSTNAME** column, then any changes would have to be made in all the appropriate records in the **ORDERS** table, as well as in the **CUSTOMERS** table.

Of course, the situation is aggravated manifold, as customer data are imported into other tables as well.

A parallel example concerns the one-to-many relationship between the **ORDERS** and the **ORDERITEMS** table (the latter stores details regarding the ordered parts). It is better to join these tables via the internal order number (the autounique key) than the user-designated order number. In that way, the user can update the order number where needed, without having to update each of the order items as well.

An autounique key should be included in any tables which contain basic elements. The reference is to data which remain relatively constant and which are likely to be imported into other tables (e.g., customers, parts, warehouses). Hence, it is included in the **CUSTOMERS**, **PART** and **WAREHOUSES** tables.

Furthermore, an autounique key should be assigned to tables whose records have a one-to-many relationship with the records of another table.

Example: As there are several order items for each order, there is an autounique key in the **ORDERS** tables.

Unique Keys

A unique key allows for rapid data retrieval via the columns that make up the key. Moreover, it ensures that no two records in a given table will have identical values in those columns. Every table must include at least one unique key.

Example: If a unique key includes **FIRSTNAME** and **LASTNAME** columns, then there will only be one record in the **EMPLOYEES** table for John Smith, although there can also be a record for John Doe or Jane Smith.

It should be emphasized that the modification of unique keys can be problematic if care is not taken. There are three ***danger zones***: adding a unique key, deleting a column from a unique key and reducing the width of a column in the unique key. In all three cases, records can be deleted inadvertently from the database.

Before adding a unique key, ensure that values have been specified for the columns which comprise it; otherwise, all existing records will have the same value in the columns comprising the unique key. As this is not permitted, all but one of the records will automatically be erased!

A similar situation may arise from the deletion of a column from a unique key. If the deletion causes more than one record to contain the same values in its unique key (which is now made up of one less column), all but one of the records in question will be deleted from the database.

Example: If the unique key contains the columns **FIRSTNAME** and **LASTNAME**, then the record for Samuel Brown is distinguished from the record for Samuel Black. If you delete the **LASTNAME** column from the key, both records will have the same value ("Samuel") in the key, and one will be deleted.

Finally, you may lose records by reducing the width of one of the columns that make up the unique key.

Example: The **CUSTDES** column, which is part of a unique key, was originally assigned a width of 12 and you reduce this to 8. If there is already data in this column, then the customers "North Stars" and "North Street" will both become "North St". Consequently, these two records would have the same unique key, and one of them would be deleted.

Nonunique key

The nonunique key is used to provide rapid access to data in the table. It should include columns which are frequently used to retrieve data and which contain highly diversified data.

Consider the difference between the following two columns in the **ORDERS** table: **CUST**, which stores a wide variety of internal customer numbers, and **ORDTYPE**, which classifies orders by type of sale. Obviously, the internal customer number distinguishes between records much better than the sale type. Moreover, it is much more likely that data will be retrieved by customer number than by sale type. In fact, you might design a query form (a sub-level of the *Customers* form, linked to it by the customer's internal number) which displays all orders for each customer. Entrance into that form would be extremely slow if **CUST** were not a nonunique key.

Note that the header of any key is automatically treated as a nonunique key by the system. In this context, "header" is defined as a group of consecutive key columns starting from the first key column. This is true of unique and nonunique keys alike. This can be understood by considering the example of a phone book. Assuming that (1) entries are sorted first by last names and then by first names, (2) no person listed has more than one entry, and (3) no

two persons listed share the same last and first names, then the unique key is comprised of two columns: **LASTNAME** and **FIRSTNAME**. So, if you know both a person's last name and first name, you can find the telephone number rapidly by skipping forward and backward. However, if you only know the last name (the header of the unique key) and the address, you can skip to this last name and then move through all entries until you find the correct address. Moreover, you can stop checking when you have finished scanning the entries with this last name. On the other hand, if you only know the person's first name and address, you would have no means of finding his or her telephone number without checking every single entry! Hence, the header of the key (**LASTNAME** in the above example) is helpful in accessing data, whereas the rest of the key, without the header, is of little use.

For a table with a unique key comprised of: internal order number (**ORD**), internal part number (**PART**) and date (**CURDATE**), both the **ORD** column and the pair **ORD/PART** would be considered nonunique. Another example is one of the nonunique keys of the **ORDERS** table, which is comprised of two columns: **CUST** and **CLOSED**. In this case, the **CUST** column is also treated as a nonunique key in its own right.

Rules for Keys

- There may only be one autounique key per table. It must comprise a single column of **INT** type which does not appear in any other of the table's keys, and it must be of first priority.
- A table must have at least one unique key.
- If there is an autounique key, it must be the first key (1) in the table and a unique key must be second. If there is no autounique key, one of the unique keys must have first priority.
- The order in which key columns are designated determines their priorities.
- The column to be included in the key must be included in the table to which the key is assigned.
- If you add a column to a key without specifying column priority, it will automatically be assigned the last available priority (e.g., if the key includes two columns, the added column will receive a priority of 3).
- Changing a key column's priority will affect the priority of the other columns in the key.
  Note: You can check the consequences of assigning key column priority by means of the SQL optimizer (see below).

Example: In a key comprised of the columns **FIRSTNAME** and **LASTNAME** in an **EMPLOYEES** table, you would give highest priority to **LASTNAME** and second priority to **FIRSTNAME**. This would allow for rapid retrieval by **LASTNAME**:
SELECT * FROM EMPLOYEES WHERE LASTNAME = 'Brown';

Options for Creating and Modifying Tables, Columns and Keys

The following options are available from the *Table Generator* menu (*System Management → Generators → Tables*). In order to run them, you must belong

to the privilege group of the superuser (*tabula*) and the PRIVUSERS system constant must be set to 1.

Note: When ***Priority*** is first installed, the *Table Generator* is invisible by default. To add it to the *Generators* menu, enter the *Menu Generator* form (menu path: *System Management* > *Generators* > *Menus*) and retrieve the *Generators* menu. Then add the **TABGEN** menu using the *Menu Items* sub-level form. For more information on adding entities to menus, see the section: **Linking the Tree to a Menu**
.

It is imperative that ***only one of these programs be run at a time***; otherwise, temporary tables will be overwritten. Thus, you must wait for one program to finish before you can run another one. Moreover, care must be taken that two individuals do not run any of these programs at the same time.

Available programs are:

- **Create Table** — To create a new table:
  1. Enter the *Define Table* form and record a table name, type (1) and title.
  2. Enter the *Columns* sub-level form and, for each table column, record a name, type, width, title and decimal precision (if needed).
  3. Enter the parallel sub-level form, *Keys*, and record the new table's keys.
  4. For each key, enter the *Key Columns* sub-level form and record the columns that make up the key.
  5. Exit all forms. The new table is created.
  6. A message appears asking if you wish to continue. To create additional tables, click *OK*; otherwise, click *Cancel*.
- **Delete Table** — You cannot delete a table if at least one of its columns appears in a form, report or procedure.
- **Change Table Name** — The revision of the name will have no effect on the forms or reports based on the table (as these forms and reports refer to the table's internal number, rather than its name). However, any SQL statements that refer to the table will have to be revised accordingly.
- **Change Table Title**
- **Add Column to Table** — To add a table column:
  1. Run the *Add Column to Table* program. The *Add Column* form opens.
  2. Choose the table to which you want to add the column.
  3. Enter the sub-level form and record the name, type, width, title and decimal precision (if needed) of the new column.
  4. Exit all forms. The new column is added to the table in question.
  5. A message appears asking if you wish to continue. To add more table columns, click *OK*; otherwise, click *Cancel*.
- **Delete Column from Table**
- **Change Column Name** — The revision will have no effect on any forms or reports that include the column (as it is the column title, rather than the column name, which appears in the user interface, and as the column is

identified in forms and reports by its internal number). However, any SQL statements that refer to the column will have to be revised accordingly.

- **Change Column Width** — Widths will be modified in any existing forms and reports that include the column. Exercise caution when reducing column width, as stored data that were originally wider than the new width will be lost (see above).
- **Change Decimal Precision** — You can only change decimal precision in the following cases (decimal precision will be modified in any existing forms and reports that include the column):
  - any real number
  - a regular integer (current precision = 0) can be changed to the value of the DECIMAL system constant
  - a shifted integer can be changed to a precision of 0 (i.e., converted to a regular integer).
- **Change Number Type** — Columns of **INT** type may be changed to **REAL** type, and vice versa, during the development phase. Caution is advised: if the column in question is already part of an SQL query (e.g., in a form trigger or **compiled** program), then certain adjustments will have to be made. For instance, the type context may have to be redefined. When a **REAL** column is converted into an **INT** column, values are rounded off to the nearest integer, and decimal precision is changed to 0 (e.g., 5.68 becomes 6 and 2.13 becomes 2). When an **INT** column is converted into a **REAL** column, two decimal places are added (i.e., a decimal precision of 2 is automatically assigned). For instance, a value of 6 becomes 6.00 and a value of 2 becomes 2.00.
  Note: Once a custom development has been installed in your working environment, this operation may fail, in which case you should use a workaround method (see **Column Types**, p. 2).
- **Change Column Title** — The revision will generally affect all forms and reports that include the column. The new title will **not** affect form columns or report columns which have been assigned a revised title in a specific form or report, as this overrides the table column title.
- **Add Key to Table** — To add a table key:
  1. Run the *Add Key to Table* program. The *Add Key* form opens.
  2. Indicate the table to which you want to add the key, the type of key (A, U or N) and the key's priority.
     Notes:
     There may only be one autounique key per table.
     The addition of a unique key to a table that already includes records should be done with caution (see above).
  3. Enter the sub-level form and record the columns in the key.
  4. Exit all forms.
  5. A message appears asking if you wish to continue. Click *OK* to add the key to the table.
  6. A second message appears asking if you wish to continue. To add more table keys, click *OK*; otherwise, click *Cancel*.
- **Delete Key from Table**
- **Change Key Priority**
- **Change Autounique to Unique**

- **Change Unique to Nonunique** — You cannot change the type of the first unique key in the table.
- **Add Column to Key** — To assign the new column a priority that has already been assigned to another key column, specify the desired priority. The new column will take over that priority and the old one (and any subsequent ones) will move down a priority.
- **Delete Column from Key** — Once a column is removed from a key, all columns with a lower priority move up a priority. On the dangers of deleting a column from a unique key, see above.
- **Change Column Priority** — The changing of a column's priority will affect the priority of the other columns in the table.

Viewing Tables in the Database

Dictionaries and Reports

In addition to enabling the construction and modification of tables, the *Tables* generator provides access to dictionaries and reports that display information on tables already in the database:

- *Table Dictionary* — Displays all tables in the database; its sub-level forms display columns, keys, and key columns.
- *Column Dictionary* — Displays attributes of all table columns.
- *Columns per Table* report
- *Keys per Table* report.

Note: In addition, various utilities accessed via the *SQL Development* program are helpful in understanding table structure. For details, see Chapter 2.

Customization Procedure

If users at a customer site work in a language other than English, you will need to install revisions in more than one language. ***Before you even begin programming*** for such a customer, enter the *System Constants* form (*System Management → System Maintenance → Constant Forms*) and change the value of the UPGTITLES constant to 0. Consequently, no titles (in any language) will be stored in the upgrade file; rather, they will be inserted into a second file (based on the upgrade file) via another program.

Note: For more on preparing upgrades for other languages, see Chapter 9.

Modifying a Standard Table

1. Make sure all users have exited the system.
2. If you add a column to the table, the column name must begin with a four-letter prefix. Use the same prefix for all table columns (as well as any other ***Priority*** entities) that you have added for a given customer.
3. When done, exit and reenter the system (this way, you will avoid a system failure to prepare any form to which you have added this table column).

Creating a New Table

- There is no need for users to exit the system
- The table name should begin with the appropriate four-letter prefix (there is no need to add this prefix to table columns).

DBI Syntax

The *Database Interpreter* (*DBI*) program comprises a database language created specially for constructing and modifying database tables.

Syntax Conventions

In delineating the syntax for the *Database Interpreter* program, the following conventions are employed:

- Anything within brackets is optional (e.g., [*filename*]). If brackets are omitted, the argument *must* be specified.
- The "|" symbol between several options indicates that only one may be chosen (e.g., [**FORMAT | DATA | ASCII**]).
- When a set of options is enclosed within curved brackets, one of the various options *must* be chosen (e.g., { *form/trigger* | *form/form_column/trigger* }).
- Characters in **bold** must be specified exactly as they appear. Characters in *italics* have to be replaced by appropriate values. An underscore between two or more italized words indicates that they should be treated as a single value (e.g., **SQLI $U**/*query_file parameter*).
- All punctuation must be designated as is (commas, colons, parentheses, and especially the semicolon that is required at the end of each SQL statement)
- Ellipses (...) indicate that several values may be indicated for the previous argument (e.g., *table_name, ...*).
- In examples, curved brackets {} enclosing text signify a comment which does not belong to the SQL statement. Comments *within* SQL statements are enclosed within a pair of slashes and asterisks (e.g., /* Open failed; no record meets condition */).

Modifying Database Tables via SQL Statements

In addition to using the options in the *Tables Generator*, you can run the *SQL Development* program and modify tables by means of SQL statements, using the syntax outlined below, provided you belong to the privilege group of the superuser (*tabula*) (for details, see Chapter 9).

Syntax for Tables

Create table

> **CREATE TABLE** *table_name* [*type*]
> *column_name1* (*type*, *width*, [*decimal_precision*,] *'title'*)
> [*column_name2* (*type*, *width*, [*decimal_precision*,] *'title'*)]
> [*column_name3* ...]

[**AUTOUNIQUE** (*column_name*)]
UNIQUE (*column_name*, ...)
[**UNIQUE** ...]
[**NONUNIQUE** (*column_name*, ...)]
[**NONUNIQUE** ...];

Delete table

DELETE TABLE *table_name*;

Change table name

FOR TABLE *table_name*
CHANGE NAME TO *new_name*;

Change table title

FOR TABLE *table_name*
CHANGE TITLE TO *'new_title'*;

Syntax for Columns

Add column to table

FOR TABLE *table_name*
INSERT *column_name* (*type*, *width*, [*decimal_precision*,] '*title*');

Delete column from table

FOR TABLE *table_name* **DELETE** *column_name*;

Change column name

FOR TABLE *table_name* **COLUMN** *column_name*
CHANGE NAME TO *new_name*;

New column width

FOR TABLE *table_name* **COLUMN** *column_name*
CHANGE WIDTH TO *integer*;

New column title

FOR TABLE *table_name* **COLUMN** *column_name*
CHANGE TITLE TO '*title*';

New decimal precision

FOR TABLE *table_name* **COLUMN** *column_name*
CHANGE DECIMAL TO *decimal_precision*;

Change number type

FOR TABLE *table_name* **COLUMN** *column_name*
CHANGE NUMBER TYPE;
/from INT to REAL and vice versa/

FOR TABLE *table_name* **COLUMN** *column_name*
CHANGE NUMBER TYPE TO REAL;
/only from INT to REAL; if already REAL, leaves as is/

FOR TABLE *table_name* **COLUMN** *column_name*
CHANGE NUMBER TYPE TO INT;
/only from REAL to INT; if already INT, leaves as is/

Syntax for Keys

Add new key to table

FOR TABLE *table_name*
INSERT { **AUTOUNIQUE** | **UNIQUE** | **NONUNIQUE** }
(*column_name*, ...)
[**WITH PRIORITY** *key_priority*];

Delete key from table

FOR TABLE *table_name*
DELETE KEY { *key_priority* **|** (*column_name 1*, ... , *column_name n*) };

Change key priority

FOR TABLE *table_name*
KEY { *key_priority* **|** (*column_name 1*, ... , *column_name n*) }
CHANGE PRIORITY TO *new_key_priority;*

Change key type from autounique to unique

FOR TABLE *table_name*
CHANGE AUTOUNIQUE TO UNIQUE;

Change key type from unique to nonunique

FOR TABLE *table_name*
KEY { *key_priority* **|** (*column_name 1*, ... , *column_name n*) }
CHANGE UNIQUE TO NONUNIQUE;

Syntax for Key Columns

Add new column to key

FOR TABLE *table_name*
KEY { *key_priority* **|** (*column_name 1*, ... , *column_name n*) }
INSERT *column_name*
[**WITH PRIORITY** *column_priority*];

Delete column from key

FOR TABLE *table_name*
KEY { *key_priority* **|** (*column_name 1*, ... , *column_name n*) }
DELETE *column_name*;

Change column priority in key

FOR TABLE *table_name*
KEY { *key_priority* **|** (*column_name 1*, ... , *column_name n*) }
COLUMN *column_name*
CHANGE PRIORITY TO *new_column_priority*;

Chapter 2: SQL Syntax

Syntax Conventions

This chapter describes syntax for SQL statements in ***Priority***, which is based on ANSI standard syntax and its interpretation, but also includes several additional features. Most of the material in this chapter applies to all SQL queries in ***Priority*** — those found in form triggers, step queries in procedures, load queries and expressions for calculated form columns and report columns. Occasionally, however, reference is made to features that only apply to one or two of these (e.g., form triggers). When this is the case, this restriction is explicitly stated.

In delineating SQL syntax, the following conventions are employed:

- Anything within brackets is optional (e.g., [*filename*]). If brackets are omitted, the argument *must* be specified.
- The "|" symbol between several options indicates that only one may be chosen (e.g., [**FORMAT | DATA | ASCII**]).
- When a set of options is enclosed within curved brackets, one of the various options *must* be chosen (e.g., { *form/trigger* | *form/form_column/trigger* }).
- Characters in **bold** must be specified exactly as they appear. Characters in *italics* have to be replaced by appropriate values. An underscore between two or more italized words indicates that they should be treated as a single value (e.g., **SQLI $U**/*query_file parameter*).
- All punctuation must be designated as is (commas, colons, parentheses, and especially the semicolon that is required at the end of each SQL statement)
- Ellipses (...) indicate that several values may be indicated for the previous argument (e.g., *table_name, ...*).
- In examples, curved brackets {} enclosing text signify a comment which does not belong to the SQL statement. Comments *within* SQL statements are enclosed within a pair of slashes and asterisks (e.g., /* Open failed; no record meets condition */).

Viewing Table Structure

The *SQL Development* (**WINDBI**) program (*System Management* → *Generators* → *Procedures*) provides access to utilities that can help you understand table structure in ***Priority***. The main utility is *Dump Table*, which creates a file containing the full definition of a designated table. The first line indicates the table name, title and type; subsequent lines delineate its columns and keys.

This is the result of the *Dump Table* program for the **ORDSTATUS** table:

```
CREATE TABLE ORDSTATUS 'Possible Statuses for Orders' 0
ORDSTATUS (INT,13,'Order Status (ID)')
ORDSTATUSDES (CHAR,12,'Order Status')
```

```
    INITSTATFLAG (CHAR,1,'Initial Status?')
    CLOSED (CHAR,1,'Close Order?')
    PAYED (CHAR,1,'Paid?')
    SORT (INT,3,'Display Order')
    OPENDOCFLAG (CHAR,1,'Allow Shipmt/ProjRep')
    CHANGEFLAG (CHAR,1,'Allow Revisions?')
    CLOSESTATFLAG (CHAR,1,'Closing Status?')
    REOPENSTATFLAG (CHAR,1,'Reopening Status?')
    INTERNETFLAG (CHAR,1,'Order from Internet?')
    OPENASSEMBLY (CHAR,1,'Open Assembly Status')
    CLOSEASSEMBLY (CHAR,1,'End Assembly Status')
    PARTIALASSEMBLY (CHAR,1,'Partial Assm. Status')
    ESTATUSDES (CHAR,16,'Status in Lang 2')
    MANAGERREPOUT (CHAR,1,'Omit from Reports')
    AUTOUNIQUE (ORDSTATUS)
    UNIQUE (ORDSTATUSDES);
```

To run the program, from the **Dump** menu, select **Table**. In the input window, designate the table you want to see. To dump more than one table at a time, separate table names with a space.

Additional utilities are accessed from the **Queries** menu (in the same window):

- **Select All** retrieves all records from a designated table.
- **Table Columns** shows the columns in a table.
- **Table Columns incl. precision** provides the same information, but also displays decimal precision.
- **Table Keys** shows the keys in a table.

Executing SQL Statements

The *SQL Development* program also provides access to the *SQL Interpreter*, which is used to execute SQL statements (from the **Execute** menu, select **SQL interpreter**). Use of the *Interpreter* requires permission, which is assigned to users in the *Authorized for SQL* column of the *Personnel File* (in the *Human Resources* menu).

In principle, any statement can be written as input to the interpreter. ***However, in order to avoid violations of referential integrity, it is highly advisable to refrain from using the Interpreter for updates and deletions of the database. As Priority provides for full automatic protection of referential integrity in its forms, such manipulations should be executed via forms or form interface programs*** (see Chapter 3 and Chapter 7). Additionally, anyone executing INSERT, UPDATE or DELETE statements must belong to the privilege group of the superuser (tabula).

Additional options for using the *SQL Interpreter* are:

- **+ optimizer** — displays the steps of data retrieval.
- **+ execution** — displays the steps of data retrieval and indicates the number of records retrieved in each step.

SQL Functions and Variables

Priority offers a set of system functions that can be used to retrieve set results (e.g., the current date and time). Furthermore, it recognizes a number of kinds of variables. These functions and variables are used in SQL statements.

Functions are identified within the SQL statement by the prefix “SQL.” Variables are identified by the prefix “:” (colon).

Each function and variable must belong to one of the following types: **CHAR, INT, REAL, DATE, TIME** or **DAY**. These are all valid column types (for an explanation of each, see Chapter 1).

System Functions

Here is a partial list of SQL functions recognized by ***Priority***.

- SQL.ENV: the current ***Priority*** company (**CHAR** type)
- SQL.USER: the internal number of the current user (**INT** type)
- SQL.GROUP: the internal user number of the group representative from whom the current user inherits privileges (**INT** type)
- SQL.DATE: the current date and time (**DATE** type)
- SQL.DATE8: the current date without time (**DATE** type)
- SQL.TIME: the current time (**TIME** type)
- SQL.DAY: the current weekday (**DAY** type)
- SQL.LINE: the line number during retrieval from the database; retrieved records are numbered consecutively (**INT** type)
- SQL.TMPFILE: a full path to a temporary file name; you can use this path to link tables to the file
- SQL.ENVLANG: the language defined for the current ***Priority*** company (in the *Companies* form)
- SQL.WEBID, SQL.CLIENTID: identification variables (**INT** type and **CHAR** type, respectively) used for ***Priority Lite*** only (see Chapter 12).
- SQL.GUID: returns a random 32-character string, using the operating system's UUID function.

Examples:
(1) You can use SQL.USER and SQL.DATE for an electronic signature.
(2) To return a numbered list of parts, use:

```
SELECT SQL.LINE, PARTNAME FROM PART FORMAT;
```

Variables

There are several types of SQL variables:

- form column variables, determined by the content of a given column in a given form
- parameter variables, determined by the content of a given parameter in a given procedure step
- user-defined variables

- system variables:
  :RETVAL — the return value of the previous query (**INT** type).
  :SCRLINE — the current form line, in triggers only (**INT** type).
  :PAR1, :PAR2 and :PAR3 — parameters for error and warning message commands (**CHAR** type); used in form triggers, step queries (in procedures) and load queries; maximum number of characters for each parameter is 64.
  :PAR4 — stores the value of the first argument in CHOOSE- triggers (**CHAR** type).
  :FORM_INTERFACE (**INT** type) — when assigned a value of 1, indicates that form records are filled by a form load interface rather than the user.
  :FORM_INTERFACE_NAME (**CHAR** type) — when the :FORM_INTERFACE variable has a value of 1, the current variable is assigned the name of the form interface in question (e.g., see BUF16 in the **LOGPART** form; on buffers, see below).
  :PREFORMQUERY (**INT** type) — assign it a value of 1 in a PRE-FORM trigger to run the trigger after each query.
  :ACTIVATEREFRESH (**INT** type) — assign it a value of 1 in a PRE-FORM trigger to refresh all retrieved records after a Direct Activation.
  :ACTIVATE_POST_FORM (**CHAR** type, width 1) — assign it a value of Y in a PRE-FORM trigger to activate the form's POST-FORM trigger upon exiting the form, even if no changes have been made (e.g., see PRE-FORM trigger in the **TRANSORDER_H** form).
  :KEYSTROKES — use in a PRE-FORM trigger to store a string containing reserved words which imitate keyboard actions.
  :HEBREWFILTER (**INT** type) — for users of ***Priority*** in Hebrew. This variable is used to ensure that Hebrew text is displayed correctly when exporting data from ***Priority*** using an SQL query. When this variable receives a value of 0, the characters in exported Hebrew text appear backwards (e.g., מילשורי טרוא ס"יב); assign it a value of 1 to ensure that such characters appear in the correct order (e.g., בי"ס אורט ירושלים).
  :HTMLACTION, :HTMLVALUE and :HTMLFIELD — used in ***Priority Lite*** procedures (see Chapter 12).
  :_IPHONE — used in ***Priority Lite*** procedures (see Chapter 12). This variable receives a value of 1 when the procedure in question is run from a mobile device (e.g., an iPhone or Android device), and a value of 0 when the procedure is run from a regular PC or iPad. By referring to this variable within the procedure itself (or one of its component reports), the procedure can be run differently depending on the type device from which it is accessed (e.g., see step 40 in the **WWWDB_PORDERS_A** procedure).
  :NOHTMLDESIGN — used in processed reports and ***Priority Lite*** procedures. When this variable receives a value of 1, reports will be produced in non-HTML format, even if HTML design options are defined for report columns. This variable is sometimes used in conjunction with the previous one (e.g., see step 40 in the **WWWDB_PORDERS_A** procedure).
  :HTMLMAXROWS — used in processed reports and ***Priority Lite*** procedures to limit the number of results that appear on the page. This is useful, for instance, if you want to run a report that displays the last 5 sales orders placed by the customer, or if you want to enable users to indicate how many results to display on a particular page of your ***Priority Lite*** website

(e.g., see step 40 in the **WWWDB_ORDERS_A** procedure).
:_CHANGECOUNT (**INT** type) — stores the number of fields in the current form record that have been revised; useful in a PRE-UPDATE or POST-UPDATE trigger when you want to perform a check or other action only if a single field, or a specific set of fields, has been changed.
:PRINTFORMAT (**INT** type) – stores the print format chosen by the user when a document is printed. Print formats are saved in the **EXTMSG** table.
:SENDOPTION (**CHAR** type) — stores the user's selection in the Print/Send Options dialogue box when a document is printed.
:ISSENDPDF (**INT** type) — when a value of 1 is received, creates a PDF document rather than an HTML document (used with :SENDOPTION).
:WANTSEDOCUMENT (**INT** type) — stores the user's selection in the *Are sent e-mails digitally signed by Outlook* column of the Mail Options dialogue box.
:EDOCUMENT (**INT** type) — used in e-documents (see page 125). When this variable receives a value of 1, sent e-documents will be synchronized with ***Priority*** and recorded as a customer task.
:GROUPPAGEBREAK (**INT** type) — used in processed reports to add a page break for the first "Group by" set (see Chapter 4, Display of Grouped Records). When this variable receives a value of 1, each group of records in the report will appear on a new page.
:FIRSTLINESFILL (**INT** type) — used in forms. When the sub-level form is entered, the variable automatically receives a value of 1; once the user runs a query in the form, the variable receives a value of 0. Useful when you want to run an automated query upon entrance into the sub-level form without restricting which records can be retrieved by a user-defined query. For example, the condition defined for the **CURDATE** column of the **CUSTNOTES** form (in the *Form Column Extension* sub-level of the *Form Columns* form) is used to retrieve all recent tasks upon first entering the form. Once these tasks have been loaded, however, users can run a query to retrieve older tasks by pressing F11.
:SQL.NET (**INT** type) — stores 0 when working in the Windows interface and 1 when working in the web interface.

Note: The following reserved words are useful with the :KEYSTROKES variable — '{Activate}N' (runs the form's Nth Direct Activation), {Exit} (executes the query), {Key Right}, {Key Left}, {Key Up}, {Key Down}, {Page Up}, {Page Down}, '{Sub-Level}N' (opens the form's Nth sub-level form), {Table/Line View} (toggles between multi-record and full-record display).

Examples:
:KEYSTROKES = '*{Exit}'; (retrieves all form records)
:KEYSTROKES = '{Key Right} 01/01/06 {Exit}' (moves one column to the right and executes a query that retrieves any records with Jan. 1, 2006, in the field).

All form column variables and parameter variables take on their respective column types. In all other cases, SQL defines variable type according to the context. That is, such variables will inherit the type of any other variable in the expression which is already defined in the form. For instance, if the **QUANT** and **PRICE** columns in the **ORDERITEMS** form are defined as **REAL**, then the

:totprice variable in the following expression is assumed to be a real number as well:

```
SELECT :ORDERITEMS.QUANT * :ORDERITEMS.PRICE INTO
:totprice
FROM DUMMY;
```

or

```
:totprice = :ORDERITEMS.QUANT * :ORDERITEMS.PRICE;
```

Similarly, a variable will inherit the type of a constant value appearing in the expression. For instance, in the expression :i + 5, *i* is assumed to be an **INT** variable.

Given an expression without a type context, SQL assigns a default variable type of **CHAR**. However, for a string with a width of 1 (single character), you must specify "'\0' +" or use a declaration for that single **CHAR** variable, for example: :SINCHAR = '\0';.

To obtain a variable of **REAL** type, create the proper type context by adding the prefix "0.0 +" to the expression. To obtain a variable of **INT**, **DATE**, **TIME** or **DAY** type, add the prefix "0 +".

For instance, assuming that the variable :j (which contains a real number) has not yet been defined, you would use the following statement to select its values:

```
SELECT 0.0 + :j FROM DUMMY FORMAT;
```

or

```
:j = 0.0;
```

If you need a variable of **REAL** type with more precision than two decimal places, initialize it as follows:

```
:CONV = 0E-9;
```

Flow Control

Several ***Priority*** commands may be used in SQL statements to affect execution flow. These are mainly used in long sequences of SQL commands (e.g., form triggers). They include:

- **GOTO** — causes a jump *forward* to a given label when the statement is successful
- **LOOP** — causes a jump *backward* to a given label when the statement is successful
- **LABEL** — signifies the place at which to continue execution of SQL statements after a GOTO or LOOP command has been encountered. The label number must be identical to that specified for the appropriate GOTO or LOOP command.
- **SLEEP** — signifies the number of seconds to pause before continuing; generally, used when waiting for a response from an external device
- **GOSUB** — causes a jump to a specific sub-routine

- **SUB** — signifies the beginning of a sub-routine; no commands from here until the next RETURN command will be executed unless specifically called by a GOSUB command
- **RETURN** — ends the sub-routine and continues with the statement following the appropriate GOSUB command
- **END** — discontinues execution of SQL statements
- **ERRMSG** — causes failure and prints out an error message on screen; used in form triggers, step queries (in procedures) and load queries
- **WRNMSG** — prints out a warning message on screen; used in form triggers, step queries and load queries
  Note: When used in a step query, this does not necessarily delay the procedure execution flow in the ***Priority*** web interface. To ensure interruption of execution flow, use the CONTINUE basic command (see **Basic Commands** in Chapter 5).
- **REFRESH** — refreshes screen with updated values after data manipulation; used only in form triggers
- **MAILMSG** — sends a message by internal mail to a user or group of users, or by external mail to one or more e-mail addresses; can include an attachment

Syntax of the Flow Control Commands

The syntax of each of these commands is as follows:

- **GOTO** *label_number* [**WHERE** *condition*];
- **LOOP** *label_number* [**WHERE** *condition*];
- **LABEL** *label_number*;
- **SLEEP** *number_of_seconds*;
- **GOSUB** *sub_number* [**WHERE** *condition*];
- **SUB** *sub_number*;
- **RETURN**;
- **END** [**WHERE** *condition*];
- **ERRMSG** *msg_number* [**WHERE** *condition*];
- **WRNMSG** *msg_number* [**WHERE** *condition*];
- **REFRESH** 1;
- **MAILMSG** *msg_number* **TO** { **USER | GROUP | EMAIL** } *recipient* [**DATA** *attachment_filename*] [**WHERE** *condition*];

Usually, the MAILMSG command retrieves the e-mail's subject and content from the message specified in the *msg_number* argument, and any file specified in the [**DATA** *attachment_filename*] option will be included as an attachment. However, you can also have the MAILMSG command create an e-mail on the basis of an existing HTML document by using a *msg_number* argument that indicates an empty message and using the [**DATA** *attachment_filename*] option to indicate an HTML attachment.

Notes:

If you use this option, the attachment indicated by the *attachment_filename* argument must be an HTML file.

If you are working on a Unicode installation, the HTML attachment should be saved in a Unicode-compliant format.

If a value has been assigned to the :_REPLYTOEMAIL variable, the MAILMSG command sends the e-mail using that value as the reply-to e-mail address. This setting overrides any default reply-to e-mail address defined for the message.

Note: This setting is applied only if you have set up external mail without Outlook.

Examples of Usage

This example illustrates the following:

- declare and open a cursor
- if the open fails, go to end
- if it succeeds, fetch the next record
- use of sub routine
- as long as there are more records to be fetched, execute the designated data manipulations on each record
- once there are no more records, close the cursor
- end execution.

```
DECLARE C CURSOR FOR ...
OPEN C;
GOTO 9 WHERE :RETVAL = 0; /* Open failed; no record meets condition */
LABEL 1; FETCH C INTO ...
GOTO 8 WHERE :RETVAL = 0; /* No more fetched records */
```

Database manipulations with the fetched fields; usually updates of some sort

```
GOSUB 100 WHERE ...;
LOOP 1;
LABEL 8;
CLOSE C;
LABEL 9;
END;
SUB 100;
```

More database manipulations with the fetched fields

```
RETURN;
```

Note: In this example the sub-routine is defined at the end. You can also define the sub-routine at the beginning of the text.

Using Sub-routines

Sub-routines are useful, for example, for loading data. The SUB command (together with an accompanying integer) marks the beginning of the sub-routine; the RETURN command signifies its end. Sub-routines are only

executed when specifically called by the appropriate GOSUB command. Thus, GOSUB 1 calls up SUB 1, GOSUB 2 calls up SUB 2, and so on. Once RETURN is encountered, execution continues from the statement following the relevant GOSUB command.

Additions and Revisions to Standard SQL Commands

Priority offers some additional features to several standard SQL commands: SELECT, ORDER BY and LIKE. Furthermore, it entails revisions to the standard SQL join.

Output Formats for SELECT

In ***Priority***, an output format command must be added to the end of a SELECT statement in order to obtain output. There are several output format commands:

- **FORMAT** — generates output comprised of column headings and data.
- **TABS** — generates data separated by tabs (\t) with titles of retrieved columns at the beginning of each record and line feed (\n) at the end of each record; useful for preparing files that can be loaded into a spreadsheet.
  Note: To create the file without titles, initialize the :NOTABSTITLE variable to 1 before executing the query.
- **DATA** — generates file structure information (header), marked as comment lines by the symbol #, as well as the data; useful for exporting ***Priority*** data to an external database.
- **ASCII** — produces output of data only (no column headings) with no spaces in between columns.
- **SQLSERVER** — same as **TABS**, except that output does not include the titles of retrieved columns.
- **ORACLE** — generates a file for **sqlldr** (SQL Loader – an Oracle utility used to load data from external files).
- **UNICODE** — generates output in Unicode (UTF-16) format.
- **ADDTO** — adds data to end of specified file, rather than replacing the contents of the file.

The syntax of these commands is:

> **SELECT** ... [{ **FORMAT | TABS | DATA | ASCII** | **SQLSERVER | ORACLE**} [**UNICODE**] [**ADDTO**] [*'filename'*]];

If a file name is specified (enclosed in single quotes), the output is dumped into that file; otherwise, it appears in standard output.

Note that the file name can be an expression. For instance,

```
SELECT * FROM PART
WHERE PART > 0
FORMAT STRCAT ('/tmp/', 'part.sav');
```

will store results in the *tmp* directory in a file named *part.sav*.

Extended LIKE Patterns

Priority includes several LIKE patterns in addition to those found in standard SQL ("_" and "%" wildcards, which represent a single character and an unlimited number of characters, respectively).

For instance, the symbol " | " may be used in pairs (as brackets) to enclose several characters or a range of characters. Any single character appearing within the brackets or falling within the range may be retrieved (e.g., **| A–D | %** yields any character or string beginning with the letter *A, B, C* or *D*, such as *Armchair, Desk, Chair*).

Moreover, the symbol "^" may be added before one or more characters enclosed in brackets, to retrieve any character *other than* those (e.g. **| ^A–D | %** yields any character or string that does *not* begin with the letter *A, B, C*, or *D*).

Finally, the delimiter "\" should be used to retrieve one of the above symbols. For instance, **A\%** yields the string *A%*.

For more examples, see the search criteria designated in the *User Interface Guide*.

Outer Join

An outer join is represented in ***Priority***'s syntax by a question mark (?) following the table ID:

```
SELECT ...
FROM FNCITEMS, FNCITEMSB ?
WHERE FNCITEMSB.FNCTRANS = FNCITEMS.FNCTRANS
AND FNCITEMSB.KLINE = FNCITEMS.KLINE;
```

As opposed to regular joins, an outer join preserves unmatched rows. Thus, if there is no join record, a null record will be retrieved from the join table and the query will succeed.

Example: The **FNCITEMS** table, which stores journal entry items, is linked to **FNCITEMSB**, a table which (among other things) stores profit/cost centers from groups 2-5. An outer join is required, as not all records in **FNCITEMS** have values in **FNCITEMSB** (that is, a given journal entry item may not necessarily include profit/cost centers from groups 2-5).

Execution Statements

Priority recognizes several commands which affect execution of the SQL statement (or group of statements).

Env

The ENV command changes the current ***Priority*** company to the specified company. The syntax is:

ENV *company*;

where the company can be defined by a string (the value appearing in the *Company* column of the *Companies* form) or a variable.

Execute

The EXECUTE command executes a specified program with designated parameters. It is mainly used in form triggers, in order to activate a program from within the form or from an SQLI step in a procedure. In any program that uses an EXECUTE command, execution occurs in the present company. It may take place in the background.

The syntax is:

EXECUTE [**BACKGROUND**] *program* [*parameter, ...*];

where the program and each of its parameters can be specified as a string or variable.

Note: For the ***Priority*** web interface, see Chapter 15.

LINK and UNLINK

The LINK mechanism creates a temporary copy of a given database table. This linked file serves a number of purposes:

- It can serve as a parameter comprised of a batch of records.
- It can function as a work area in which data manipulation is performed prior to report output. Results are then sent for display in the processed report.
- It is used by form load interfaces.

The LINK command is complemented by the UNLINK command. The syntax of each is as follows:

LINK *table_name1* [ID] [**TO** *filename*1];

{ database manipulations }

UNLINK [**AND REMOVE**] *table_name1* [ID];]

The LINK command ties a designated table to a temporary file having an identical structure, including all the columns and keys from the original table. If the linked file does not yet exist, this command creates it. If it does exist, the command simply executes the linkage. If you specify a file name, this file can be used later on.

Example: The SQLI program can create a linked file in one procedure step, whose contents are used in a report in the next step. For details, see Chapter 5.

The linked file is initially empty of data. All subsequent operations that refer to the original table are actually executed upon that temporary file, until the UNLINK command is encountered.

You cannot link the same table more than once prior to an UNLINK. If you do, the second (and any subsequent) LINK to that table will return a value of –1 (i.e., that particular query will fail, but the rest of the queries will continue to

be executed). However, you can circumvent this restriction by adding a different suffix (table ID) to the table name for each link.

Example: While you cannot link the **ORDERS** table twice, you can link both **ORDERS A** and **ORDERS B**. In this case, you will obtain another copy of the table for each link, and these may be used as separate files. Then, after linking, you could perform the following query:

```
INSERT INTO ORDERS A
SELECT *
FROM ORDERS B
WHERE ...;
```

After the database manipulations are completed and the required data is stored in the linked file, you can simply display the results in a processed report, without affecting the database table. The UNLINK command stores the temporary file in the specified (linked) file and undoes the link. All succeeding operations will be performed on the original table. If the AND SET option is used, then the copy of the table, with all its data manipulations, will be stored in the original table. All operations that succeed the UNLINK command will be performed on the database table and not on the copy.

Use the AND REMOVE option if you wish the linked file to be deleted when it is unlinked. This is necessary when working with loops, particularly when manipulations are carried out on the data in the linked file. If you do not remove the linked file, and the function using LINK and UNLINK is called more than once, you will receive the *same copy* of the table during the next link. So, if you want the LINK command to open a new (updated) copy of the table, use UNLINK AND REMOVE.

Important! Working with a linked file can be dangerous when the link fails. If the query is meant to insert or update records in the linked table and the link fails, then ***everything is going to be executed on the real table***! Therefore, you must either include an ERRMSG command for when the link fails, or use the GOTO command so as to skip the part of the query that uses the linked table.

For example:

```
SELECT SQL.TMPFILE INTO :TMPFILE FROM DUMMY;
LINK ORDERS TO :TMPFILE;
ERRMSG 1 WHERE :RETVAL <= 0;
/*database manipulation on the temporary ORDERS table */
UNLINK ORDERS;
```

or:

```
SELECT SQL.TMPFILE INTO :TMPFILE FROM DUMMY;
LINK ORDERS TO :TMPFILE;
GOTO 99 WHERE :RETVAL <= 0;
/*database manipulation on the temporary ORDERS table */
UNLINK ORDERS;
LABEL 99;
```

Return Values and Statement Failure

The following table displays return values for each SQL statement and delineates when each statement fails.

Command	Return Values	Failure When...
DECLARE	1 (success)	never
OPEN	number of records 0 upon failure	too many open cursors (>100), including recursive opens no selected records
CLOSE	1 upon success 0 upon failure	cursor is not open
FETCH	1 if fetched 0 if end of cursor	cursor is not open no more records in cursor
SELECT	number of selected records 0 upon failure	no record met WHERE condition
SELECT … INTO	1 upon success 0 upon failure	no record met WHERE condition
INSERT … SELECT	number of inserted records –1 if no record meets WHERE condition	no record met WHERE condition there were selected records, but none was inserted (generally due to unique key constraint or insufficient privileges)
INSERT_VALUES	1 upon success 0 upon failure	failed to insert
UPDATE … WHERE CURRENT OF *cursor_name*	1 upon success 0 upon failure	cursor is not open no more records in cursor there is a record, but it was not updated
UPDATE	number of updated records 0 upon failure to update –1 if no record meets WHERE condition	no record met WHERE condition there were selected records, but none was updated (generally due to unique key constraint or insufficient privileges)
DELETE … WHERE CURRENT OF *cursor_name*	1 upon success 0 upon failure	cursor is not open no more records in cursor there is a record, but it was not deleted
DELETE	number of deleted records 0 upon failure to delete –1 if no record meets WHERE condition	no record met WHERE condition there were selected records, but none was deleted (generally due to unique key constraint or insufficient privileges)
RUN	returns what the query returns	
ENV	1 upon success 0 upon failure	
EXECUTE	PID of the child process	
LINK	2 if new file has been created 1 if link to existing file 0 upon failure to link –1 if more than one link to same table name	
UNLINK	1 upon success 0 upon failure	
GOTO		no such label found forwards

Command	Return Values	Failure When...
LOOP		no such label found backwards
LABEL		never
END		never

Non-standard Scalar Expressions

In addition to standard SQL scalar expressions, ***Priority*** also recognizes several other expressions. These are described below. Moreover, it offers support of bitwise operations on integers.

Note: To test the SELECT statements below, you can record them in the window of the *SQL Development Program*, as follows, and then execute them, using the *SQL Interpreter*.

Conditional Expression

Following C language, ***Priority*** uses the symbols ? : to designate a conditional expression *(if ... then ... else ...)*. The syntax of a conditional expression is:

(*expression* **?** *expression* **:** *expression*)

If the first expression yields a True value, then the second will determine the resulting value. If not, then the third expression will determine that value.

Example: A conditional expression could be used for a calculated column in the **ORDERSBYCUST** report which warns that the order is overdue:
(SQL.DATE8 > ORDERITEMS.DUEDATE AND ORDERITEMS.BALANCE > 0 ? '*' : ' ')
That is, if the current date is later than the due date and the balance to be delivered is greater than 0, then display an asterisk; otherwise, leave blank.

Numbers

The following expressions are related to numbers:

- **ROUND**(*m*) — rounds *m* (a real number) to the nearest integer and treats it as an integer

  SELECT ROUND(1.45) FROM DUMMY FORMAT; /* 1 */

- **ROUNDR**(*m*) — rounds *m* (a real number) to the nearest integer but treats it as a real number

  SELECT ROUNDR(1.45) FROM DUMMY FORMAT; /* 1.000000 */

- **MINOP**(*m, n*) — returns the minimum value between two numbers

  SELECT MINOP(1.5,2) FROM DUMMY FORMAT; /* 1.500000 */

- **MAXOP**(*m, n*) — returns the maximum value between two numbers

  SELECT MAXOP(1.5,2) FROM DUMMY FORMAT; /* 2.000000 */

- **EXP**(*m, n*) — treats *n* as an exponent of *m*

```
SELECT EXP(3,2) FROM DUMMY FORMAT; /* 9 */
SELECT EXP(2,3) FROM DUMMY FORMAT; /* 8 */
```

- **SQRT**(*m*) — returns the square root of *m* rounded to the nearest integer, where *m* is an integer

```
SELECT SQRT(10) FROM DUMMY FORMAT; /* 3 */
```

- **SQRTR**(*m*) — returns the square root of *m*, where *m* is a real number

```
SELECT SQRTR(10.0) FROM DUMMY FORMAT; /* 3.162278 */
```

- **ABS**(*m*) — returns the absolute value of *m*, where *m* is an integer

```
SELECT ABS(-5) FROM DUMMY FORMAT; /* 5 */
```

- **ABSR**(*m*) — returns the absolute value of *m*, where *m* is a real number

```
SELECT ABSR(-5.3) FROM DUMMY FORMAT; /* 5.300000 */
```

- *n* **MOD** *m* — calculates modular arithmetic

```
SELECT 10 MOD 4 FROM DUMMY FORMAT; /* 2 */
```

You can also use the **MOD** function to retrieve the time from a DATE 14 variable:

```
SELECT 17/05/09 12:25 MOD 24:00 FROM DUMMY FORMAT; /*
12:25 */
```

- **REALQUANT**(*m*) — inputs a shifted integer and translates it to a real number, where the number of places that the decimal point is moved is determined by the value of the DECIMAL system constant (usually, 3). Used in reports to define a calculated column that, for example, displays *Quantity* x *Price*, when *Quantity* is a shifted integer and *Price* is a real number.

```
:ORDERITEMS.TQUANT = 1000;
SELECT REALQUANT(:ORDERITEMS.TQUANT) FROM DUMMY
FORMAT;
/* 1.000000 assuming the DECIMAL constant = 3 */
```

- **INTQUANT**(*m*) — inputs a real number and translates it to a shifted integer, where the number of places that the decimal point is moved is determined by the DECIMAL system constant. Used in the form interface (*Form Load Designer*) when the load table is **GENERALLOAD**, and you want to use one of the table columns for quantity as a shifted integer.

```
SELECT INTQUANT(1.0) FROM DUMMY FORMAT; /* 1000 assuming
the DECIMAL constant = 3 */
```

- **ITOH**(*m*) — returns the hexadecimal value of *m*, where *m* is an integer

```
SELECT ITOH(10) FROM DUMMY FORMAT; /* a */
```

- **HTOI**(*m*) — inputs a hexadecimal value and translates it to its corresponding integer

```
SELECT HTOI(b) FROM DUMMY FORMAT; /* 11 */
```

Strings

The following expressions are related to strings:

- **ITOA**(*m, n*) — outputs *m* as a string having *n* characters, where both values are integers (leading zeroes are added where necessary)

Note: If no *n* is specified, or if the value of *n* is less than what is needed, the minimum required width will be used.

```
SELECT ITOA(35,4) FROM DUMMY FORMAT; /* '0035' */
SELECT ITOA(35) FROM DUMMY FORMAT; /* '35' */
```

- **ATOI**(*string*) — outputs the designated string as an integer

```
SELECT ATOI('35') FROM DUMMY FORMAT; /* 35 */
```

- **RTOA**(*m, n, USECOMMA*) — outputs *m* (a real number) as a string, displaying *n* decimal places according to the decimal format for the current language.

Note: If *USECOMMA* is not included, the decimal format 1,234.56 will be used.

```
SELECT RTOA(150654.665,2,USECOMMA) FROM DUMMY
FORMAT;
/* '150.654,67' assuming decimal format is 1.234,56 */

SELECT RTOA(3.665432,2) FROM DUMMY FORMAT; /* '3.67' */
```

- **STRLEN**(*string*) — outputs the length of the string (an integer)

```
SELECT STRLEN('Priority') FROM DUMMY FORMAT; /* 8 */
```

- **STRCAT**(*string1*, *string2*, ...) — outputs the concatenation of given strings

---

Note: The length of the resulting concatenation is limited to 127 characters.

```
SELECT STRCAT('abc','ba') FROM DUMMY FORMAT; /* 'abcba' */
```

- **STRIND**(*string, m, n*) — beginning from the *m*th position in a given string, retrieves *n* characters, where *m* and *n* are fixed values

```
SELECT STRIND('Priority',3,2) FROM DUMMY FORMAT; /* 'io' */
```

- **SUBSTR**(*string, m, n*) — beginning from the *m*th position in a given string, retrieves *n* characters, whether *m* and *n* are variables or fixed values

```
:STR = ‘Priority’;
:I = 3;
:T = 2;
SELECT SUBSTR(:STR, :I, :T) FROM DUMMY FORMAT; /* 'io' */

SELECT SUBSTR('Priority',3,2) FROM DUMMY FORMAT; /* 'io' */
```

- **RSTRIND**(*string, m, n*) — same as STRIND, except that the string is read from right to left

```
SELECT RSTRIND('Priority',3,2) FROM DUMMY FORMAT; /* 'ri' */
```

- **RSUBSTR**(*string, m, n*) — same as SUBSTR, except that the string is read from right to left

```
:STR = 'Priority';
:I = 3;
:T = 2;
SELECT RSUBSTR(:STR, :I, :T) FROM DUMMY FORMAT; /* 'ri' */

SELECT RSUBSTR('Priority',3,2) FROM DUMMY FORMAT; /* 'ri' */
```

- **STRPREFIX**(*string, n*) — retrieves the first *n* characters of the string, where *n* is a fixed value

```
SELECT STRPREFIX('Priority',2) FROM DUMMY FORMAT; /* 'Pr' */
```

- **STRPIECE**(*string, delimiter, m, n*) — for a given input string and delimiter (which breaks up the string into parts), retrieves *n* parts, beginning from the *m*th part

---

Note: The string and parameters *m* and *n* may be variables, but the delimiter must be a fixed value.

```
SELECT STRPIECE('a/b.c.d/e.f','.',2,1) FROM DUMMY FORMAT; /* 'c'
*/
SELECT STRPIECE('a/b.c.d/e.f','/',2,1) FROM DUMMY FORMAT; /*
'b.c.d' */
SELECT STRPIECE('a/b.c.d/e.f','.',1,3) FROM DUMMY FORMAT; /*
'a/b.c.d/e' */
SELECT STRPIECE('a/b.c.d/e.f','/',1,3) FROM DUMMY FORMAT; /*
'a/b.c.d/e.f' */
```

- **ISALPHA**(*string*) — indicates whether a given string begins with a letter and is comprised solely of: uppercase and lowercase letters, digits, and/or _ (underline); yields 1 if it is, 0 if it is not.

```
SELECT ISALPHA('Priority_21') FROM DUMMY FORMAT; /* 1 */
SELECT ISALPHA('21Priority') FROM DUMMY FORMAT; /* 0 */
```

- **ISPREFIX**(*string1*, *string2*) — indicates whether the first string is the prefix appearing in the second string

```
SELECT ISPREFIX('HEEE','HEEE_ORDERS') FROM DUMMY
FORMAT; /* 1 */
SELECT ISPREFIX('HEEWE','HEEE_ORDERS') FROM DUMMY
FORMAT; /* 0 */
```

- **ISNUMERIC**(*string*) — indicates whether a given string is comprised solely of digits; yields 1 if it is, 0 if it is not. Useful when you wish to ensure that a given column of **CHAR** type is made up only of digits (i.e., a zip code).

Note: You would not use **INT** type in this case, because you do not want the value to be treated like a number (i.e., you want the zip code to appear as 07666 and not as 7,666).

```
SELECT ISNUMERIC('07666') FROM DUMMY FORMAT; /* 1 */
SELECT ISNUMERIC('14.5') FROM DUMMY FORMAT; /* 0 */
```

- **ISFLOAT**(*string*) — indicates whether a given string is a real number; yields 1 if it is, 0 if it is not.

```
SELECT ISFLOAT('14.5') FROM DUMMY FORMAT; /* 1 */
```

- **TOUPPER**(*string*) — changes characters to uppercase letters

```
:LOW = 'marianne';
SELECT TOUPPER(:LOW) FROM DUMMY FORMAT; /* MARIANNE
*/
```

- **TOLOWER**(*string*) — changes characters to lowercase letters

```
:UPPER = 'MARIANNE';
SELECT TOLOWER(:UPPER) FROM DUMMY FORMAT; /* marianne
*/
```

- **ENTMESSAGE**(*entity_name, entity_type, message_number* — returns the message for *message_number* of entity *entity_name* with type *entity_type*.

```
SELECT ENTMESSAGE('ORDERS','F',3) FROM DUMMY FORMAT;
/* You cannot revise the number of an itemized order. */
```

Dates

Dates, times and days are stored in the database as integers. Dates may be represented to the user in American (MMDDYY) or European (DDMMYY) format, depending on the type assigned to the language being used (in the *Languages* form: *System Management → Dictionaries → Translation*).

The following examples are in ***American date format***.

- **DAY**(*date*) — yields the number of the weekday on which the specified date falls (where Sun=1, Mon=2, etc.).

```
SELECT DAY(03/22/06) FROM DUMMY FORMAT; /* 4 */
```

Note: This number can then be translated into the name of the weekday by means of the **DAYS** table (for the application's base language of English) and the **LANGDAYS** table (for any additional languages). These tables store the names of all days in the week in what ever language you are using.

- **MDAY**(*date*) — yields the number of the day in the month

```
SELECT MDAY(03/22/06) FROM DUMMY FORMAT; /* 22 */
```

- **WEEK**(*date*) — yields an integer comprised of the year (last one or two digits of the year) and the number of the week in the year (two digits, between 01 and 53)

```
SELECT WEEK(03/22/06) FROM DUMMY FORMAT; /* 612 */
```

- **WEEK6**(*date*) — yields an integer comprised of the year in 4 digits and the number of the week in the year (two digits, between 01 and 53)

```
SELECT WEEK6(03/22/06) FROM DUMMY FORMAT; /* 200612 */
```

- **MWEEK**(*week*) — given a value for week (the last two digits of a year and the number of a week in that year), yields the number of the month in which that week falls

```
SELECT MWEEK(0612) FROM DUMMY FORMAT; /* 3 */
```

- **MONTH**(*date*) — yields the number of the month in the year

```
SELECT MONTH(03/22/06) FROM DUMMY FORMAT; /* 3 */
```

- **QUARTER**(*date*) — yields a string comprised of the annual quarter in which the date falls followed by the four digits of the year

```
SELECT QUARTER(09/22/06) FROM DUMMY FORMAT; /* 3Q-2006
*/
```

- **YEAR**(*date*) — yields an integer comprised of the four digits of the year

```
SELECT YEAR(03/22/06) FROM DUMMY FORMAT; /* 2006 */
```

- **TIMELOCAL**(*date*) — yields the number of seconds from January 1, 1970, to the specified date

```
SELECT TIMELOCAL(05/04/06) FROM DUMMY FORMAT; /*
1146693600 */
```

- **CTIME**(*int*) — yields the date corresponding to the given number of seconds since January 1, 1970 02:00

```
SELECT CTIME(1146693600) FROM DUMMY FORMAT;
/* Thu May 04 01:00:00 2006 */
```

- **BEGINOFMONTH**(*date*) — yields the date of the first day of the month

```
SELECT BEGINOFMONTH(05/04/06) FROM DUMMY FORMAT; /*
05/01/06 */
```

- **BEGINOFQUARTER**(*date*) — yields the date of the first day of the quarter

```
SELECT BEGINOFQUARTER(05/04/06) FROM DUMMY FORMAT; /*
04/01/06 */
```

- **BEGINOFHALF**(*date*) — yields the date of the first day of the six-month period (half a year) in which the date falls

```
SELECT BEGINOFHALF(10/22/06) FROM DUMMY FORMAT; /*
07/01/06 */
```

- **BEGINOFYEAR**(*date*) — yields the date of the first day of the year

```
SELECT BEGINOFYEAR(10/22/06) FROM DUMMY FORMAT; /*
01/01/06 */
```

- **ENDOFMONTH**(*date*) — yields the date of the end of the month

```
SELECT ENDOFMONTH(04/22/06) FROM DUMMY FORMAT; /*
04/30/06 */
```

- **ENDOFQUARTER**(*date*) — yields the date of the end of the quarter

```
SELECT ENDOFQUARTER(03/22/06) FROM DUMMY FORMAT; /*
03/31/06 */
```

- **ENDOFHALF**(*date*) — yields the date of the end of the half-year

```
SELECT ENDOFHALF(03/22/06) FROM DUMMY FORMAT; /*
06/30/06 */
```

- **ENDOFYEAR**(*date*) — yields the date of the end of the year

```
SELECT ENDOFYEAR(03/22/06) FROM DUMMY FORMAT; /*
12/31/06 */
```

- **ATOD**(*date, pattern*) — converts dates, times and days into internal numbers (mainly used to import external data)

  see usage below

- **DTOA**(*date, pattern*) — converts dates, times and days in the system to ASCII (mainly used to print out or display data to the user)

  see usage below

Pattern Components for ATOD and DTOA Expressions

The following pattern components can be included in ATOD and DTOA expressions (those marked with an asterisk (*) only apply to DTOA). Of course, more than one component can be used in the same expression.

Note: You can add punctuation marks (e.g., dashes, slashes, commas) and spaces between pattern components as desired.

- **MMM** or **mmm** — abbreviated form (first three letters) of month name (Jan)
- **MMMM** or **mmmm** — full name of the month (January)
- **MONTH** — abbreviated form of month name and the last two digits of the year (Jun-06)
- **MM** — number of the month (01)
- **DD** — date in the month (15)
- **YY** — last two digits of year (06)
- **YYYY** — all four digits of year (2006)
- **day*** — weekday (Mon)
- **hh:mm** — hours and minutes (12:05)
- **XX/XX/XX** — date with two-digit year, displayed in American or European format, depending on the language type defined in the *Languages* form.
- **XX/XX/XXXX** — date with four-digit year, displayed in American or European format, depending on the language type defined in the *Languages* form.
- **FULLDATE*** — the month name (abbreviated form), date and four-digit year

Converting a String to a Date: Examples

```
SELECT ATOD('06/21/06','MM/DD/YY') FROM DUMMY FORMAT;
/* 06/21/06 (June 21, 2006, in American format) */
```

```
SELECT ATOD('06/21/2006','MM/DD/YYYY') FROM DUMMY
FORMAT;
/* 06/21/06 (June 21, 2006, in American format) */

SELECT ATOD('062106','MMDDYY') FROM DUMMY FORMAT;
/* 06/21/06 (June 21, 2006, in American format) */

SELECT ATOD('311006','DDMMYY') FROM DUMMY FORMAT;
/* 31/10/06 (October 31, 2006, in European format) */

SELECT ATOD('31102006','DDMMYYYY') FROM DUMMY FORMAT;
/* 31/10/06 (October 31, 2006, in European format) */
```

Converting a Date to a String: Examples

Unless otherwise stated, examples are in American date format.

```
:DATE = 06/01/06; /* June 1, 2006 */

SELECT DTOA(:DATE,'MMMM') FROM DUMMY FORMAT; /* June */

SELECT DTOA(:DATE,'MMM') FROM DUMMY FORMAT; /* Jun */

SELECT DTOA(:DATE,'MM') FROM DUMMY FORMAT; /* 06 */

SELECT DTOA(:DATE,'MONTH') FROM DUMMY FORMAT; /* Jun-06
*/

SELECT DTOA(:DATE,'day') FROM DUMMY FORMAT; /* Thu */

SELECT DTOA(06/01/06,'XX/XX/XX') FROM DUMMY FORMAT;
/* 06/01/06 (June 1, 2006, in American format; January 6, 2006, in
European) */

SELECT DTOA(:DATE,'FULLDATE') AS 'FULLDATE' FROM DUMMY
FORMAT;
/* Jun 01,2006 */

:DATE = 06/01/06 12:33;

SELECT DTOA(:DATE,'MM/DD/YY hh:mm,day') FROM DUMMY
FORMAT;
/* 06/01/06 12:33,Thu */

SELECT DTOA(:DATE,'MMM-YY') FROM DUMMY FORMAT; /* Jun-
06 */

SELECT DTOA(:DATE,'MMMM-YYYY') FROM DUMMY FORMAT;
/* June-2006 */

SELECT DTOA(:DATE, 'The current date is MM-DD-YY, and the time
is hh:mm.') FROM DUMMY FORMAT;
```

Chapter 3: Forms

Introduction

Forms are constructed and modified in the *Form Generator* form and its sub-levels (*System Management* → *Generators* → *Forms*). They serve a variety of purposes:

- They insert records into the database.
- They retrieve records from the database.
- They allow for the updating of retrieved records.

All three functions can be simultaneously served by the very same form. Moreover, you can construct a read-only query form in virtually the same manner that you design an updateable form.

Forms can be seen as windows into specific database tables. In this context, there are two types of tables: the form's base table and its join tables. Each form is derived from a single base table. Data stored in any columns of that table can be viewed and modified in the form. Moreover, new entries in the form will add records to the base table. In contrast, join tables contain imported data — i.e., data that can be displayed in the form, but not modified there. A given form can display data from several different tables; therefore, it can have any number of join tables.

A form is characterized by:

- a unique name
- a title
- a base table
- a set of form columns derived from the columns of the base table
- additional form columns imported from other tables
- calculated columns whose values are determined by other columns; their values are not stored in any table
- a set of triggers, which execute commands during form use.

Form Attributes

►To open a new form and record its attributes, use the appropriate columns in the *Form Generator* form.

Form Name

The form name is a short name by which the form is identified by the system. As this name is used in SQL variables in form triggers, there are certain restrictions (which also apply to form column names):

- Only alphanumeric values (uppercase and lowercase letters and digits) and the underline sign may be used (no spaces).
- The name must begin with a letter.

- You may not use a reserved word (a list of reserved words appears in the *Reserved Words* form — *System Management → Dictionaries*).
- The name assigned to any newly created forms must include a common four-letter prefix (the same one you use for all entities that you add to ***Priority*** for the customer in question; e.g., **ACME_ORDERS**).

Notes:

It is advisable to assign the form the same name as its base table.

Because SQL variables are based on the form name, any changes in the name must be accompanied by changes in the appropriate SQL statements.

Form Title

The form title is the means of identifying the form in the user interface. The designated title will appear in menus and at the top of the form when it appears on screen. Changes in form titles do not affect triggers in any way.

Base Table

Each form is derived from a single table, known as its base table. The form serves two interrelated purposes. First, it acts as a window into the table, displaying the data stored there. Second, it updates the base table whenever records are added to, deleted from or revised in the form. The form inherits its columns (including their names, widths, titles and types) from the base table.

Note: When creating a new form, you should also create a new base table for it (see Chapter 1).

Application

Each form is assigned an application, which is used to classify forms by the type of data they access (e.g., FNC for the Financials module). When creating a new form, specify a code word that aids in retrieval.

Note: The application code can be used to define a target form to be reached from a given form column (see below).

Module

Each form belongs to a given ***Priority*** module. As different modules are included in each type of ***Priority*** package, users are restricted to those forms whose modules that have purchased. When creating a new form, specify “Internal Development.” This way you (and your customers) will be able to use your own forms no matter which modules of ***Priority*** have been purchased.

Query Forms

A query form is one in which the user is not permitted to add, modify or delete records. Such a form is meant solely to display information retrieved from the database.

Example: *Warehouse Balances* (**WARHSBAL**) is a query form that displays balances per part for the warehouse in question, calculated automatically on the basis of recorded warehouse transactions.

It is not necessary to flag any of the columns in a query form as read-only. In fact, it is advisable to make the columns in the unique key updateable for the user's convenience.

Tip: To create a form in which records cannot be inserted or updated, but deletions are allowed, assign read-only status to all the form's columns (see below).

Blocking Record Deletion

Sometimes you wish to restrict user operations in a given form to inserts and updates, disallowing record deletions.

Example: Forms in which constants are defined do not allow for record deletion.

Blocking Definition of a Multi-Company Form

To prevent users from defining a given form as a multi-company form, specify *x* in the *One-to-many* column.

When Creating a New Form

When you are designing your own form, once the above attributes have been designated, the form may be prepared as an executable file and loaded on screen, and values may be filled in its columns. All built-in triggers (discussed later in this chapter) will be activated.

The result, of course, will be very rough. All columns (including internal numbers) will be displayed and updateable. Form column position, which determines the order of form column display, will be based on the order in which columns were assigned to the table. Thus, it is likely that you will want to make modifications. And you will probably wish to import data from other tables. In addition, you will sometimes need to establish one-to-many relationships (between an upper-level form and its sub-levels). Finally, you may wish to create your own triggers.

Note: See rules for customizing forms, at the end of this chapter.

Form Column Attributes

►To record attributes for form columns, use the appropriate columns in the sub-level *Form Columns* form.

Column Names and Titles

The form column name serves to identify the form column in the system and is used mainly in SQL variables in form triggers. Consequently, it is subject to certain restrictions (which also apply to form names):

- Only alphanumeric values (uppercase and lowercase letters and digits) and the underline sign may be used (no spaces)
- The column name must begin with a letter.
- You may not use a reserved word.
- Any new column must begin with the appropriate four-letter prefix.
- There can be no more than 330 columns in a form.

Any change in the form column's name will require appropriate changes in SQL variables which refer to that column.

The designated name can be identical to the name of the table column from which the form column is derived. However, two different form columns which are derived from the same table column must be given different names.

Example: The *Warehouse Transfer* form (**DOCUMENTS_T**) refers to two different warehouses (sending warehouse and receiving warehouse). Though they are both derived from the same table column (**WARHSNAME**), they must have different form column names. Hence, the form column referring to the sending warehouse is **WARHSNAME**, while that referring to the receiving warehouse is **TOWARHSNAME**.

The form column title is utilized in the user interface. That is, it appears as a column heading in the form itself. The title is automatically inherited from the relevant table column. It may, however, be revised (even translated into another language). Moreover, when a form is screen-painted (see below), the title appearing in the screen-painter is displayed instead.

Example: In the **DOCUMENTS_T** form, both **WARHSNAME** and **TOWARHSNAME** inherit the title *Warehouse Number* from their common table column. To distinguish between them in the form, their titles have been changed to *Sending Warehouse* and *Receiving Warehouse*, respectively.

Order of Column Display

The order in which columns appear in the form is determined by their relative position (an integer). Integers determining column position need not be consecutive. The column assigned the lowest integer appears first, that with the next highest integer appears second, and so on.

Obviously, it is unnecessary to be concerned with the position of hidden columns (see below). Nonetheless, it is helpful to move these columns out of the way, by assigning them all the same high integer (e.g., 99).

Notes:

Positions have no effect on screen-painted forms; the designer can place each column wherever desired.

Users can apply form design (*Organize Fields*) to affect how columns are displayed specifically for them.

Hidden Columns

Not all form columns ought to be displayed. For instance, there is generally no reason to display the internal number of the table's autounique key. You should also hide the internal numbers through which data are imported from other tables.

Example: **ORD** and **CUST** are internal columns hidden from the **ORDERS** form.

The *Order Items* form (**ORDERITEMS**) exemplifies two other instances in which data are not displayed. In addition to the **PART** column (through which data are imported from the **PART** table), the **ORD** and **LINE** columns are hidden. The former is used to link the form to its upper-level (**ORDERS**), establishing a one-to-many relationship. The latter is used to sort the order items (during data retrieval) according to the order in which they were originally specified; its value is determined by a trigger. Neither of these columns need to be displayed in the form.

Tip: Have the system manager use the *Organize Fields* utility to hide columns from individual users.

Mandatory Columns

Some of the displayed columns in the form must be filled in; others are optional. For instance, the user is required to indicate the date of each sales order, whereas specification of a price quotation is optional. Similarly, the user cannot place an order without specifying a customer number, but a sales rep number would not be necessary. Therefore, the **CURDATE** and **CUSTNAME** columns are mandatory.

When a column is mandatory, built-in triggers will not allow the user to leave the line without specifying data for this column.

Note: Whenever a form load interface is used to update a form, that interface must fill in all mandatory columns. If it does not, the **INTERFACE** program will fail (see Chapter 7).

The columns that make up the base table's unique key are a special case: they must always contain data (either filled in automatically by the system or manually by the user), regardless of whether they are flagged as mandatory. If data are missing from any of these columns, the record will be incomplete and the built-in triggers will not allow the user to leave the line. It is therefore recommended that, when the user is supposed to fill in the needed data, you flag the column(s) in question as mandatory. In this way, users will know in

advance that data must be designated here. Otherwise, they will not find out until they try to leave the line and receive an error message.

Note: The *Privilege Explorer* can be used to make columns mandatory for specific users. Run the *Privilege Explorer Wizard* for details.

Read-only Columns

Not all displayed form columns should be updatable. Usually, columns whose values are determined automatically (generally by a trigger) should be read-only. For instance, the **QPRICE** column of the **ORDERS** form is determined by the sum of the prices of its order items (as designated in the sub-level **ORDERITEMS** form).

Another example is the **AGENTDES** column. Owing to built-in fill triggers, the sales rep name is filled in automatically once the sales rep number is specified in the **ORDERS** form. This is because (1) the rep's name is imported from the **AGENTS** table, and (2) the rep's number (**AGENTCODE**) is a unique key in that table.

Notes:

If you want the entire form to be read-only, you can make it a query form, thereby blocking all record insertions, updates and deletions (see above).

The *Privilege Explorer* can be used to make updateable columns read-only for specific users. Run the *Privilege Explorer Wizard* for details.

Balances: Special Read-only Columns

Priority offers the option of a ***balance*** column — a special read-only column that displays financial balances. In order to distinguish between credit and debit balances, one is displayed within parentheses while the other appears in regular format. By default, it is the debit balances that are displayed in parentheses, both in forms and in reports. If you want credit balances to be displayed that way instead, use the CREDITBAL system constant. First add it to the **SYSCONST** table using an INSERT statement and then assign it a value of 1.

You can also display a ***cumulative balance*** in a form. To do so, add two columns to the form:

- A balance column — Specify *B* in the *Read/Mandatory/Bal* column. This must be a numerical column (**INT** or **REAL**); its value will be added to the cumulative balance column.
- A cumulative balance column — Specify *A* in the *Read/Mandatory/Bal* column. In the *Form Column Extension* form, record the type (**INT** or **REAL**), and in the *Expression/Condition* column, record 0 or 0.0, respectively.

In addition, you can include an ***opening balance*** column, if you want cumulative balance calculations to start from something other than 0. In that case, specify *C* in the *Read/Mandatory/Bal* column.

Example: See the **BAL_BASE, BAL** and **OPENNBAL** columns in the **ACCFNCITEMS** form.

Boolean Columns

In order for a form column to be defined as Boolean, it must be of **CHAR** type and have a width of 1. When a Boolean column is flagged, the value of that field in the table is *Y*; when it is blank, the value in the table is *'\0'*. Users see a box which they can flag or leave empty (just like the one you use to define Boolean columns).

Attachment Columns

Priority offers the option of an attachment column — a special form column that is used to attach a file to the current record. Such columns are displayed together with a paper clip icon, which can be used to open the Windows Explorer and navigate to the file in question. Alternatively, users can record the file path manually. An example of this type of column is the *File Name* column of the *Customer Documents for Task* form, a sub-level of the *Tasks* form.

In order for a form column to be defined as an attachment column, it must be of **CHAR** type and the form column name must contain the string **EXTFILENAME** (e.g., **PRIV_ EXTFILENAME**).

Address Column

Another option is to allow users to open Google Maps from within a form. For example, by selecting *Map* (in the Windows interface, from *Tools* in the top menu; in the web interface, from *Run*) from the address column in the *Customers* form, the user can open Google Maps to see the customer's location.

In order for a form to have the option of opening a map, it must contain a special column named **ADDRESSMAP**. You can add this column to a customized form using your standard prefix (**XXXX_ADDRESSMAP**).

Note: It is not possible to add such a column to a standard form.

Special Date Columns

As mentioned earlier, dates are stored in ***Priority*** as integers, which correspond to the number of minutes elapsed since Jan. 1, 1988 (for example, Dec. 31, 1987 = -1440). Hence, the date 01/01/1988 is stored as "0". Since ***Priority*** forms do not generally display zero values, this date is not displayed in form columns. If you want the value 01/01/1988 to be displayed in a particular column, the form column name must contain the string **BIRTHDATE** (e.g., the **BIRTHDATE10** column in the **USERSB_ONE** form).

Note: When a new record is inserted into the database and the date column is empty, the system assigns a default value of 0 (i.e., 01/01/1988). If you use the above technique to display the date 01/01/1988, this value will appear in the date column of all such records. In this case, you may want to define another trigger, such that when a new record is inserted into the database and the date column in question is empty, the column receives a different default value (e.g., 01/01/1900).

Sorting Data

Priority allows you to control the default order of records appearing in a given form. That is, you may assign sort priorities to one or more columns, and you may designate the type of sort.

A given form column's sort priority (an integer) determines how records are sorted on screen. The lower the integer assigned to a given column, the higher its priority in a sort.

In addition to sort priority, you can also indicate the type of sort. There are four options:

- ascending (the default sort)
- descending
- alphanumeric ascending
- alphanumeric descending.

An alphanumeric sort operates strictly according to the ASCII table. Thus, A13 will come before A2 and B200 will precede B39 (in ascending order). In contrast, the regular sort treats consecutive digits as a *number* (rather than individual characters). Hence, in ascending order, A2 precedes A13 and B39 precedes B200.

Note: The designated sort priority and type constitute the default sort. A different sort order may be imposed by the user during data retrieval (see the *User Interface Guide*).

Imported Data

In addition to the columns derived from its base table, a form can also display information that is stored in another table. This table is known as a join table. A form can have several join tables from which it imports data.

Example: The **ORDERS** form imports data from several tables, among them **CUSTOMERS** and **AGENTS**. Thus, the customer number and the sales rep (among other things) have been added to the form.

To add a column, specify a form column name, abiding by the above-mentioned restrictions (e.g., **XXXX_CUSTNAME**). The table to which that column belongs (e.g., **CUSTOMERS**) will be filled in automatically, unless the same column name appears in more than one table. In that case, designate the table name yourself. Next, assign the new form column a position. Finally, flag the imported columns as read-only or mandatory where desired. In general, all imported columns except for those belonging to the join table's unique key should be read-only, as their values are automatically filled in by built-in fill triggers.

There is one exception to the above: when the data imported from another table is updated in the form. In such a case, you do not specify a column name (or table name) in this form, nor do you designate a join column and join table (see below). Instead you need to use the *Form Column Extension* sub-

level form, in a similar manner as you record calculated columns. For more details, see the section on “Calculated Columns” (below).

Example: See the **STARTDATE** column in the *Service Calls* form (**DOCUMENTS_Q**).

Join Columns

The customer who places a given order is stored in the **ORDERS** table solely by means of the internal customer number (**CUST**). This internal number is derived from the **CUST** column in the **CUSTOMERS** table.

When an order is recorded in the *Sales Orders* form, the number of the customer who placed that order must be specified as well. ***Priority*** then uses that number to obtain the customer’s internal number. The internal number is then stored in the appropriate record in the **ORDERS** table.

Let’s say, for instance, that the user records Order 1000 for customer P600. As the internal number assigned to this customer is 123, this internal number will be stored for Order 1000. This connection — or join — between the internal customer number in the **ORDERS** table and the internal customer number in the **CUSTOMERS** table is not automatic. It must be explicitly specified in the *Form Columns* form. You must indicate the column by which the join is made (*Join Column*) and the table to which it belongs (*Join Table*).

There are certain restrictions on the column(s) through which the join is executed:

- These columns must appear in the join table’s unique (or autounique) key.
- They must include all key columns.

Note: The column comprising an autounique key meets both conditions required of a join column.

Special Joins

There are two special types of joins:

- multiple joins — where imported form columns are derived from the same table column
- outer joins — that allow for unmatched rows between the base and join tables.

A good example of a ***multiple join*** is found in the **DOCUMENTS_T** form, based on the **DOCUMENTS** table, which includes two different warehouses: the sending warehouse and the receiving one. The **DOCUMENTS** table stores the internal numbers of the sending and receiving warehouses in the **WARHS** and **TOWARHS** columns, respectively. Both of these internal numbers are joined to the same join column: **WARHS** from the **WAREHOUSES** table. That is, the join is made to the same table twice — once to import data regarding the sending warehouse, and again to import data with respect to the receiving warehouse. A distinction is made between the two joins by means of the *Join ID*: the join for the sending warehouse is assigned a *Join ID* of 0, while the join for the receiving warehouse is assigned a *Join ID* of 1.

Similarly, a distinction must be made between the form columns that are imported through each join. For instance, the numbers of both these warehouses (**WARHSNAME** and **TOWARHSNAME**, respectively) are imported from the same table column: **WARHSNAME** from the **WAREHOUSES** table. It is essential that the number of the sending warehouse be imported through join 0, whereas the number of the receiving warehouse be imported through join 1. Thus, the former is assigned a *Column ID* of 0, whereas the latter is assigned a *Column ID* of 1. Any other value imported from the **WAREHOUSES** table (e.g., the warehouse description, the bin number) is likewise assigned the appropriate *Column ID*.

Important note: When creating your own multiple joins, use a join ID and column ID greater than 5.

As opposed to regular joins, an ***outer join*** allows for unmatched rows between the base and join tables. To designate the outer join, add a question mark (?) in the relevant *Column ID* or *Join ID* column, next to the number of the ID. The decision as to where to put the question mark (column ID? join ID?) depends on where the null record is expected to be encountered. If it is in the table from which the form column is derived (i.e., the one appearing in the *Table Name* column of the *Form Columns* form), then add the question mark to the column ID. If the null record is expected to appear in the join table, attach the question mark to the join ID. In the case of an additional join between the outer join table and another table, the question mark should appear in ***each*** of these join IDs.

Example: The **FNCITEMS** table, which stores journal entry items, is linked to **FNCITEMSB**, a table that (among other things) stores profit/cost centers from groups 2-5. An outer join is required, as not all records in **FNCITEMS** have values in **FNCITEMSB** (that is, a given journal entry item may not necessarily include profit/cost centers from groups 2-5). Moreover, as access to the **COSTCENTERS** table is via **FNCITEMSB**, it is necessary to create an outer join there as well (for instance, via the **COSTC2** column).

Note: Outer-joined tables are accessed after regular join tables.

Calculated Columns

In addition to form columns derived from the base table, as well as form columns imported from other tables, you can also create columns which display data derived from other form columns. These data are not stored in or retrieved from any database table. They are filled in when the form row is exited. The value of a calculated column is determined on the basis of other columns in the form, as defined in an expression.

Example: The **ORDERITEMS** form includes the **VATPRICE** column, which displays the line item's extended price after tax is added. The value of this column (of **REAL** type) is determined by the part price as well as the percentage of tax that applies to the part in question.

Finally, an imported column that is updated in the form is created just like a calculated column. In this case, the expression does not calculate the column value, but rather defines the join table and join column.

To add a calculated column to a given form, take the following steps:

1. Designate a unique column name in the *Form Column Name* column of the *Form Columns* form (abiding by all column name restrictions; see above).
2. Specify its position within the form in the *Pos* column.
3. If a calculated column displays a value that should not be revised, specify *R* in the *Read-only/Mandatory* column. Leave the column blank in the case of an imported column that is updated in the form (e.g., the **STARTDATE** column in the **DOCUMENTS_Q** form) or when the value in the calculated column updates a value in another column (e.g., the **BOOLCLOSED** column in the **ORDERS** form).
4. Designate the column's width in the *Width* column. In the case of a real number or a shifted integer, designate decimal precision as well.
5. Specify the column title in the *Revised Title* column. Unlike regular form columns, which inherit titles from their respective table columns, calculated columns have to be assigned titles. If you forget to do so, the column will remain untitled when the form is accessed by a user.
6. Enter the sub-level form, *Form Column Extension*.
7. Write the expression that determines the value of the column in the *Expression/Condition* column, using SQL syntax (see Chapter 2). If there is not enough room for the entire expression, continue it in the sub-level form.
8. Designate the column type (e.g., **CHAR, INT, REAL**) in the *Column Type* column of the *Form Column Extension* form. Note that the *Type* column in the *Form Columns* form will ***not*** display the type of a calculated column.

Note: Once you exit the sub-level form, a check mark will appear in the *Expression/Condition* column of the *Form Columns* form. This flag helps you to find any calculated columns in the form at a glance.

Sometimes you may wish to include a column that sets a condition for the entire record. In that case, do not create a calculated column, as described above. Instead, add a dummy column to the form (column name = **DUMMY**; table name = **DUMMY**) and assign ***it*** the desired condition. The condition itself must be preceded by "**=**1 AND."

Example: See the **DUMMY** column in the **FNCIFORQIV** form.

Custom Columns: Data Authorization

Priority's Data Authorization mechanism enables you to restrict the content that users can access, based on attributes such as the branch in which a document was recorded and/or the sales rep who performed the transaction.

If you have created a custom table column for a piece of data that is normally subject to data authorizations, you will need to apply the same Data Authorization settings defined for the standard table column to the custom table column as well.

In order for the custom column to inherit data privileges defined for an existing system column, the two columns need to be linked. To do so:

1. Enter the *Data Privileges* form (*System Management → System Maintenance → Privileges → Auxiliary Programs (Privileges) → Data Privileges*).
2. Add a line for the custom table column, specifying the name of the custom *Column* and the *Table* in which it appears, and the names of the *Main Column* and *Main Table* to which it should be linked. Consequently, the custom *Column* inherits the data privileges defined for the *Main Column*.

Example: If data privileges have been defined for the *Main Column* **BRANCH** in the *Main Table* **BRANCHES**, you can add a line to the *Data Privileges* form for a second branch column **BRANCH2** in the **ORDERS** table. As a result, the **BRANCH2** column inherits the data privileges defined for the *Main Column* (**BRANCH**).

To grant data authorization to a given user, add one or more lines to the **USERCLMNPRIV** table, as follows:

1. Record the relevant username in the **USER** column.
2. In the **COLUMNA** column, specify the name of the column for which data privileges have been defined, as recorded in the **MAINCOLUMN** column of the **CLMNPRIV** table.
3. Do **one** of the following:
   - If the user in question is authorized to see all data, insert a single line for this user in the **USERCLMNPRIV** table and record '*' in the **VALUE** column.
   - If the user in question is authorized to see specific data only, insert a separate line in the **USERCLMNPRIV** table for each value in the designated column for which the user is authorized.

Sub-level Forms

Priority forms are grouped within logical contexts into a tree-like configuration, representing one-to-many or one-to-one relationships. The root of the tree is the form which is accessed directly from a menu (e.g., *Sales Orders*); that is, it has no upper-level forms of its own. The branches of the tree are the sub-level forms of the root form, their sub-levels and so on. Any given form can have several sub-levels on the same level.

Example: *Order Items* and *Tasks* are both sub-levels of the *Sales Orders* form.

Relationships Between Upper- and Sub-level Forms

The relationship between an upper-level form and its sub-level may be:

- *one-to-many* — multiple records in the sub-level form are linked to a single record in the upper-level form
- *one-to-one* — a single record in the sub-level is linked to the record in the upper-level form.

Generally, the relationship between forms is one-to-many. For instance, each sales order can have several different order items. Sometimes, however, you wish to limit the sub-level form to a single record.

►To obtain a one-to-one relationship, specify *N* in the **One-to-Many** column of the *Form Generator* form at the record for the sub-level form.

Linking Upper-level and Sub-level Forms

A sub-level form is used to display data that are relevant for a single record in its upper-level form. This linkage between the upper and sub-level form is executed through one or more columns. If the upper-level form has an autounique key, the link is made through the column comprising that key. Otherwise, the link is made through each column of its unique key.

Example: It is important to ensure that the parts ordered in Order 1000 are linked to the record of that order in the *Sales Orders* form, while the parts ordered in Order 1007 are linked to the record of that order. Thus, the **ORD** column in the *Order Items* form is equated with the **ORD** column in the *Sales Orders* form.

The linkage is created by means of a condition written for the relevant column in the sub-level form (e.g., the **ORD** column in the *Order Items* form).

In the case of an ***updatable*** sub-level (like *Order Items*), use the following format:

:*formname***.***columnname*

That is, begin with a colon, followed by the name of the *upper-level* form, a period, and finally the name of the column in the upper-level form to which the linkage is made (e.g., :ORDERS.ORD).

If the sub-level is a ***query*** form (like *Warehouse Balances*), add an equal sign to the beginning of the condition:

=:*formname***.***columnname*

The addition of the equal sign allows users to delete records from the upper-level form even though records appear in the sub-level.

►Specify this condition in the *Expression/Condition* column of the *Form Column Extension* form, a sub-level of the *Form Columns* form.

Notes:

You can use the double dollar sign ($$) as a wildcard in place of the upper-level form name (e.g., :$$.ORD). Moreover, if the link is made between columns having identical names, you can use the @ sign as a wildcard in place of the form column name (e.g., :$$.@).

A detailed explanation of conditions is given later.

Creating a Form Tree

In addition to linking a sub-level form to its upper-level via their common record, you also have to locate them within a form tree. As a given upper-level form can have several sub-levels, you need to determine the order in which sub-levels appear (e.g., in the list of sub-levels). Each sub-level form's position is defined by an integer: the lower the integer, the higher the position. Assign the lowest integer to the most frequently used sub-level form, which will then serve as the default sub-level. Integers need not be consecutive.

►To link a sub-level form to its upper-level form and to define its position, use either the *Sub-level Forms* form or the *Upper-level Forms* form. Both are sub-levels of the *Form Generator* form.

Linking the Tree to a Menu

In addition, the root form of a form tree should be linked to a menu, from which it will be loaded (alternatively, the root form can be activated from within another form via Direct Activation; see below). As a menu generally contains several items, you are required to indicate the position of the root form within the menu (specify an integer). Again, the lower the integer, the higher the position of the form in relation to other items appearing in the menu.

Linkage between a root form and its menu is stored in the **MENU** table.

►To link a root form to a menu and to define its position, use either the *Menu/Form Link* form, a sub-level of the *Form Generator* form, or the *Menu Items* form, a sub-level of the *Menu Generator* form.

Conditions of Record Display and Insertion

Priority allows for two types of condition:

- a simple query condition, which restrict the records from the base table that can be retrieved into the form
- an Assign condition, which both restricts record display in the form and assigns the designated value to new records.

►To specify a condition (regardless of type), use the *Expression/Condition* column of the *Form Column Extension* form (a sub-level of *Form Columns*). If it is too long to fit in that column, continue in the sub-level form.

Once the *Form Column Extension* form is exited, a check mark appears in the *Expression/Condition* column of its upper-level form, *Form Columns*. This flag makes it easy to find any form column with a condition.

Query Condition — Record Display

Query conditions restrict the records from the base table that can be accessed by the form in question. The condition must begin with a comparative operator (<, >, <=, >=, <>, =). Only records that comply with the prescribed condition will appear in the form.

Of course, users can add their own query conditions from within the form. Yet, whereas user stipulations change for each data retrieval (in keeping with the current situation), the predefined query conditions will always apply.

Example: To restrict the display of records in the **WARHSBAL** form to parts with actual balances in the warehouse, the following condition would be written for the **TBALANCE** column: >0. This condition should be written in any form that displays such balances. Hence, while the **WARHSBAL** table will include records with a balance of 0, only those records which meet the condition will appear in each of the forms.

As mentioned earlier, you should also use a simple query condition to link a query sub-level to its upper-level form. For instance, the condition linking a balance to its respective warehouse (=:WAREHOUSES.WARHS) restricts displayed balances to those for this particular warehouse. Without this restriction, balances for all warehouses would appear.

You include the operator (=) in this condition so that the user can delete records from the upper-level form even though records appear in the sub-level. The reason for this usage is that the built-in delete triggers do not allow the deletion of any records in an upper-level form when there are records with ***assigned values*** in the sub-level form. As opposed to Assign conditions (see below), which assign values into each record in the sub-level, a simple query condition does not. Hence, deletion will not be blocked.

Assign Condition — Record Display and Insertion

An Assign condition is distinguished from a simple query condition in that no comparative operator is used. This type of condition not only restricts record display in the form to those records that hold a certain value, but also assigns that value to each new record.

You are already familiar with one Assign condition — the one that links an updatable sub-level form to its upper-level (e.g., ORDERS.ORD). This condition, written in this case for the **ORD** column of the **ORDERITEMS** form, has a two-fold effect:

- It assigns an internal value (the value of **ORD** in the **ORDERS** form) to the **ORD** column of the **ORDERITEMS** form, whenever a new record is added.
- It restricts record display to those ordered items which contain that value in their **ORD** column.

Example: All order items in order 0998 will receive a value of “13” in their **ORD** column, whereas all items in order 1010 will receive an internal value of “22.” Consequently, the former set of ordered goods will appear in the **ORDERITEMS** form under order 0998, while the latter set will appear under order 1010.

Another use of an Assign condition is to distinguish between data records that are stored in the same table but displayed in different forms.

Example: Consider the following forms, all based on the **DOCUMENTS** table: *Customer Shipments* (**DOCUMENTS_D**), *Goods Receiving Vouchers* (**DOCUMENTS_P**) and *Warehouse Transfers* (**DOCUMENTS_T**). The records in these three forms are distinguished by their type (*D* for shipment, *P* for GRV and *T* for transfer). Thus, for example, the condition ‘D’ is written for the **TYPE** column of the **DOCUMENTS_D** form. Consequently, whenever a new record is inserted in the form, the hidden **TYPE** column is automatically assigned a value of *D*. Moreover, data retrieval will only retrieve records from the **DOCUMENTS** table that have a type of *D*. Similar conditions are written for the **TYPE** columns of **DOCUMENTS_P** and **DOCUMENTS_T** (*P* and *T*, respectively).

Direct Activations

While ***Priority*** entities are generally accessed from the menus, they may also be activated from within a specific record in a given form. The form’s Direct Activations may include:

- root form — the activation loads the form and its sub-levels;
- report — runs the report, then allows for display, printout, etc.
- procedure — if the procedure includes a program, the program will be run; if it entails processing of a report, the report will be run.

During activation of a program, report or procedure, input is usually based on the content of the form line on which the cursor rests (sometimes user input is also allowed; see Chapter 5 for details). Upon exiting the entity that was activated, the user returns to his or her original place in the form.

Example: Select *Order Confirmation* from the list of Direct Activations in the *Sales Orders* form, and a printout will be created for the sales order on which the cursor rests in the form.

Direct activations can be executed in the foreground or the background. If they are in the background, the user will be able to continue work in the form while the program/procedure is being executed or while the report is run, processed (where necessary) and printed. If direct activation is in the foreground, the user has to wait until it is completed to continue work in the form; and if the current form record (i.e., the line on which the cursor rests) has been updated during the direct activation, any new values will automatically be displayed. Use background execution to run large programs and to print out reports. Designate foreground activation when an entity is executed quickly and sends a response to the waiting user.

►To create the link between a form and an entity to be directly activated, retrieve the form from the *Form Generator* form and specify the entity in the *Direct Activations* sub-level form. Or enter the generator for the entity (e.g., the *Report Generator* for a report) and specify the form from which it is activated in the *Menu/Form Links* sub-level form.

Note: For more details on direct activation of simple reports, see Chapter 4. For more on processed reports and other procedures, see Chapter 5.

Form Refresh

Generally, the data retrieved in any given form are relatively static. That is, changes are usually made by one user at a time. Sometimes, however, the data displayed in a form are highly dynamic — updated periodically by the system. In the former case, it is enough to retrieve records once; in the latter, a periodic refresh of the form is in order.

If you want a form to be refreshed periodically by the system, you need to fill in the *Refresh Form* form, a sub-level of the *Form Generator* form. Indicate the number of seconds that should pass without user input between one form refresh and the next. Also indicate whether all existing records should be retrieved during the refresh (this is important if new records have been added since the last update) or only those that were retrieved previously. The form refresh works per node on the form tree. That is, it only affects the specific form for which it is designated; it does not affect any sub-levels. As the automatic refresh involves access to the server, it should be used sparingly and with caution. When it is employed, the TIMEOUT constant is disabled.

Additional ways of forcing a form refresh are:

- Include the REFRESH command in a form trigger.
- To refresh all retrieved records following a Direct Activation, include :ACTIVATEREFRESH = 1 in the PRE-FORM trigger of the form in question.

Accessing a Related Form

When an imported column appears in a given form, the user can move from that column to a target form (generally the form which is derived from the join table). In this way, the user can gain easy and rapid access to the target form without having to use the menus.

The Target Form

The target form must meet two conditions:

- it must be a root form (have no upper-level form of its own).
- its base table must include the column from which the user originated.

The move from an imported column to a target form is automatic whenever the target form and its base table share the same name (the form's title is irrelevant). As only one form in the entire application meets that condition (form names are unique), that form is considered the default target form.

However, you may also wish to designate a target form manually. Generally, this is the case when several application forms share the same base table. Obviously, only one of those forms have the same name as that table. Therefore, if you wish the user to access a different form, you have to do ***one*** of the following:

- make the form in question the main target (record *M* in the *Zoom/International* column of the *Form Generator* form); it can be reached from any form in the application;
- make it an application target (record *Z* in the same column); it will only be reached from forms having the same application code; or
- make it a specific target form for a single form column (in the *Target Form Name* column of the *Form Column Extension* form); this will only affect the move from the column in question.

All manually designated targets always override the default target form. The target form designated for a specific column overrides all other target forms. The application target always overrides the main target.

Note: To disable automatic access from a given column, specify the **NULL** form as the target form in the *Form Column Extension* form.

Sometimes the table column name in the target form differs from the table column name in the current form. In such a case, the names of both columns must be entered into the **ZOOMCOLUMNS** table, so as to indicate which column in the target form is equivalent to the current one.

Example: The **LOADLOCATIONSMIG** form includes the **SONPARTNAME** column, for which the **LOGPART** form is specified as the target form. In order for the correct record to be retrieved when entering **LOGPART**, the following is recorded in the *Target Zoom Columns* (**ZOOMCOLUMNS**) form: *Source Column* = SONPARTNAME; *Target Column* = PARTNAME.

Dynamic Access

Sometimes you want the target form to vary, based on the data displayed in a given record. For example, in the *Audit Trail* form (**LOGFILE**), the target form of the **LOGDOCNO** column (*Doc/Invoice Number*) is the relevant document (**DOCUMENT_D, DOCUMENTS_N, AINVOICES**, etc.).

This is achieved by defining a special hidden form column, **ZOOM1**, as well as specifying **ZOOM1** as the target form in the *Form Column Extension* sub-level form. This hidden form column holds the internal number (in the current example, **EXEC**) of the relevant form for each record. The **DOCTYPES** table has a special **EXEC** column that holds the internal number of the form for that document type. In the PRE-FORM trigger of the **LOGFILE** form, the variables that hold internal numbers of the relevant forms are initialized. In some cases the target form is the form defined in the **DOCTYPES** table; in other cases the target form is one of the variables initialized in the PRE-FORM trigger of the **LOGFILE** form.

Creating a Text Form

A text form is a one-column form of **CHAR** type which allows the user to write unlimited comments applicable to a given record in a given form. ***Priority*** provides a program that enables you to automatically construct text forms (and their corresponding tables) and link them to the proper upper-level form. The text form will inherit the name of the upper-level form, with the addition of the characters **TEXT**, as well as its application code. Moreover, it will receive the title *Remarks.* You may, of course, revise any of these (in the *Form Generator* form).

Note: The newly created table will inherit the name of the base table for the upper-level form, with the addition of the characters **TEXT**, as well as its table type.

►To automatically create a text form and link it to its upper-level form, run the *Create Text Form* program (*System Management → Generators → Forms*).

The newly created text form will appear in the *Form Generator* form, and its linkage to the upper-level form will be displayed through the appropriate forms (*Sub-level Forms* and *Upper-level Forms*). Its base table will appear in the *Table Dictionary*.

Note: See also **Text Form Variables** (p. 60).

Removing HTML Tags from a Text Table

All text tables in ***Priority*** (e.g., **PARTTEXT**, **ORDERSTEXT**) contain HTML tags. Sometimes, however, you may want to receive the content of these tables without the HTML tags — for example, when exporting ***Priority*** data to an external database. You can use the **DELHTML** compiled program to delete HTML tags from any text table that has the structure defined below.

Important Note: Do **not** run this program on the original table. Instead, create a linked table on which the program can run, so that the original table will not be affected.

You can run the **DELHTML** program on any text table that is composed of the following columns and unique key:

Table Columns:

1. **IDCOLUMN1**
2. **IDCOLUMN2**
3. **IDCOLUMN3**
4. **IDCOLUMN4**
5. **TEXT**
6. **TEXTORD**
7. **TEXTLINE**

Unique Key:

8. **IDCOLUMN1**
9. **IDCOLUMN2**
10. **IDCOLUMN3**
11. **IDCOLUMN4**
12. **TEXTLINE**

Note: **IDCOLUMN1** - **IDCOLUMN4** refer to identifying columns included in the table's unique key, such as **PART**, **ORD**, **ORDI**.
IDCOLUMN2, **IDCOLUMN3**, and **IDCOLUMN4** apply to tables whose unique key comprises more than two columns, such as **USEREDUCATIONTEXT**.

The **DELHTML** program receives a table name and a linked table as input. Its output is the linked table in which all HTML tags have been removed from the **TEXT** column.

Example: To delete HTML tags from the **PARTTEXT** table for Part ‘010’, the following code would be used:

```
:PART = 0;
SELECT PART INTO :PART FROM PART WHERE PARTNAME = '010';
SELECT SQL.TMPFILE INTO :TXT FROM DUMMY;
LINK PARTTEXT TO :TXT;
GOTO 99 WHERE :RETVAL <= 0;
INSERT INTO PARTTEXT SELECT * FROM PARTTEXT ORIG
WHERE PART = :PART AND TEXT <> '';
/* Don't insert empty lines into the link table */
UNLINK PARTTEXT;
/* text with HTML tags */
SELECT TEXT FROM PARTTEXT WHERE PART = :PART FORMAT;
EXECUTE DELHTML 'PARTTEXT', :TXT;
LINK PARTTEXT TO :TXT;
GOTO 99 WHERE :RETVAL <= 0;
/* same text without HTML tags */
SELECT TEXT FROM PARTTEXT WHERE PART = :PART FORMAT;
UNLINK PARTTEXT;
LABEL 99;
```

Designing a Screen-Painted Form

What is a Screen-Painted Form?

Most of the frequently used upper-level forms are screen-painted. That is, they display a single record organized into sets of vertical columns within tabs. The number of tabs, their titles, how columns are organized and the like, are all determined by the designer. Users can then use the *Organize Field* utility to rearrange tabs and their columns, hide columns, change titles and so forth.

Note: To the user, any form in full-record display format looks screen-painted. The difference is that, in a form that is not screen-painted, tabs and their contents are created automatically.

Screen-painted forms are created by the *Screen Painter* program, which can be run from the *Forms* menu or (by Direct Activation) from the *Form Generator*. Each screen-painted form is stored in a file in the *system\document\lang.*XX directory (where XX is the language code, as defined in the *Languages* form). The filename is dNNNN.scr, where NNNN is the form's internal number (the value of its autounique key).

Tip: To find the internal number of a given form, run the following query via the *SQL Development* program (fill in the form name):
SELECT EXEC FROM EXEC
WHERE ENAME = *formname* AND TYPE = 'F' FORMAT;

Once a user runs the *Organize Fields* utility, he or she views a "private" version of the screen-painted form. Thereafter, most changes you make in the *Screen Painter* will not be visible (with the exception of added columns, which will appear in the **Misc** tab — for more details, see below). To remove the "private" version of the screen-painted form, the user must select *Restore Defaults* (within the *Organize Fields* window).

Creating Your Own Screen-Painted Forms

Before you begin to design your own forms, you should look at existing forms in the *Screen Painter* (e.g., **ORDERS**, **CUSTOMERS**). It may also be helpful to copy an existing file of a screen-painted form to a new file (whose name includes the internal number of the form you are designing) and then make revisions via the *Screen Painter*.

The tool itself is relatively intuitive. The following are some basic guidelines for using it:

- To begin, from the **Form** menu, select **Open Release** and record the internal name of the form in question. A blank screen should appear.
- To set the background color, from the **View** menu, select **Paper Color**.
- Use the **Draw** menu (or the respective icons) to create needed geometric shapes:
  - Use a large rectangle for the form itself.
  - Use smaller ones to hold sets of form columns.
  - Use a round rectangle to hold the form title. To add the title within the rectangle, record it in the Text Properties window that opens automatically.
  - Click **Select Tool** (or press **Esc**) when done.
- Right-click on a rectangle to change font, font size, font color, line color, fill color, etc.
- To insert form columns:
  - Click the **Import Form Column** icon (or select **Column** from the **Draw** menu).
  - Select the desired column.
  - Click and drag to define the column's location.
  - The column and its title appear.
  - Repeat to import another column.
  - Click **Select Tool** (or press **Esc**) when done.

- To add a form tab:
  - Click the **Insert a Tab** icon (or select **Tab** from the **Draw** menu).
  - Click and drag to define its size.
  - Assign a tab title in the **Text Properties** window.
  - Repeat to add another tab.
  - Click **Select Tool** (or press **Esc**) when done.
- To revise a title, double-click on it and make the necessary changes in the **Text Properties** window.
- To align a set of titles and/or columns, select them all (click while holding down the **Shift** key) and, from the **Format** menu, select **Align** and then the desired direction (e.g., **Left**).
- To distribute a set of titles and/or columns evenly, lining them up one under the other, select them all and, from the **Format** menu, select **Spacing**, then **Vertical**, then specify 0.
- To obtain shapes (e.g., rectangles) of equal sizes, select them both and, from the **Format** menu, select **Size** and then the desired value (e.g., **Tallest**).
- To move objects, select and drag. By default, movement is tied to the grid; to free it temporarily, from the **View** menu, select **Snap to Grid** (so it is ***not*** flagged) and make any needed adjustments. When done, reselect it (so it is flagged again).
- To add free text (such as the form title), click the **T** icon; to include a picture (like in the **LOGPART** form), click the **P** icon.
- To rearrange tabs or insert a tab in the middle:
  - Make sure the screen (with the grid) is open to full size.
  - Move all tabs, beginning with the last one and ending at the point that you want to make the revision, down to the grid area.
  - Insert a new tab, as needed.
  - Return the tabs to the large rectangle, in the desired order.
- To save all changes, click the **Save Form Release** icon. It is recommended that you do so regularly, as there is no undo tool.

Rules for Creating a Screen-Painted Form

The following rules must be followed in order to obtain proper results in the form interface:

- Each tab must be comprised of a set of column titles next to a set of columns.
- Each set of columns must be located within a single rectangle (the rectangle must be added to the tab).
- You cannot include more than two sets of titles + columns in a single tab.
- Each set of columns and each set of column titles must be properly aligned (use the **Align** tool described above).

Revising a Standard Screen-Painted Form

It is not necessary to use the *Screen Painter* when you add customized columns to a standard form. Any added column will automatically be included in the **Misc** tab when the user opens the form. The user can then use the

Organize Fields utility to move it to another tab. If the **Misc** tab gets filled up to capacity, any additional columns will not be accessible. In such a case, the user must move the columns appearing in the **Misc** tab to another tab and then return to the **Misc** tab to see the additional columns.

If you use the *Form Generator* to hide a column that appeared in an existing screen-painted form, users will continue to see the column title, but will see an empty box in place of the column data. In such a case, they should use the *Organize Fields* utility to hide the column in question.

Creating a "Totals" Form

A "Totals" form is a sub-level of a given document's itemization form. It is a query form used to display selected columns (usually prices) from the root form, as well as text recorded for the document in question.

Example: The **ORDERSTOTAL** form, which displays sales order totals, is a sub-level of **ORDERITEMS**, which in turn is a sub-level of the root form, **ORDERS**.

- To create a "Totals" form:
  - Use the base table of the root form (e.g., ORDERS) as the base table of the "Totals" form.
  - Include all columns that you wish to display in the form.
  - Make the "Totals" form the first sub-level of the root form's first sub-level.
- To display the document's text:
  - Add a hidden **CHAR** column to the "Totals" form with a width of 20.
  - In the *Expression/Condition* of that column, record the name of the form from which the text should be displayed (e.g., 'ORDERSTEXT').

Note: Use the *Screen Painter* (see above) to design the form. To display the document text, add an "Html Viewer" element:

Form Triggers

A form trigger in ***Priority*** is a set of SQL statements that affect record insertions, updates and deletions during form completion. Before creating triggers, it is important to fully understand the use of SQL syntax in ***Priority*** (see Chapter 2).

Priority provides for several types of triggers in the construction of forms. It both includes built-in triggers of its own and enables you to create your own triggers: Column (or Field) triggers, Row triggers and Form triggers. This

section first explains variables that are employed in triggers, then briefly illustrates ***Priority***'s built-in triggers, and finally provides an in-depth look at user-designed triggers. Each type of user-designed trigger is described and exemplified, in addition to which a series of complex examples are presented later on.

It should be noted that the term *form trigger* is used in two senses:

- generically, to refer to all triggers activated by either exiting a form column, exiting the row or entering/exiting the form;
- to refer to a specific type of trigger — PRE-FORM and POST-FORM — that takes effect when the form is entered or exited, respectively.

To distinguish between these two usages, the former (generic sense) will be called a "form trigger," whereas the latter (specific sense) will be called a "Form trigger."

On form debugging, see Chapter 8.

SQL Variables

Priority's form preparation mechanism recognizes the following as SQL variables in form triggers; anything else will generate a warning message:

- a form column variable — e.g., :ORDERS.QUANT (see below)
- any variable preceded by a semi-colon, which does not include a period — e.g., :QUANT
- system variables.

Form Column Variables

Priority defines three SQL variables for each form column:

- ***:form_name.form_column_name*** — stores the column's current value on screen (note the required semi-colon and period)
- ***:form_name1.form_column_name*** — stores its value in the table (note the addition of "1" before the period)
- ***:form_name.form_column_name*.TITLE** — stores the form column title.

Thus, :ORDERITEMS.QUANT refers to the ordered quantity currently designated in the **ORDERITEMS** form. In contrast, :ORDERITEMS1.QUANT refers to the ordered quantity stored in the **ORDERITEMS** table. If you are updating an existing record and have not left the current line in the form, then these variables will hold two different values — the updated value and the previous one, respectively. Along the same lines, :ORDERITEMS.QUANT.TITLE refers to the title of this column. This is useful, for instance, as a parameter in error or warning messages for triggers (see below). These form column variables may be used in creating expressions and triggers for the form.

Wildcards

Generally, expressions and triggers refer to the current form (e.g., **ORDERITEMS**) or to its upper-level form (e.g., **ORDERS**). ***Priority*** allows you to use the dollar sign as a wildcard in place of these form names. Use one dollar

sign ($) for the current form, two ($$) for the next level up, three ($$$) for the next level, and so on.

Consider the following trigger in the **ORDERITEMS** form, which computes the *Extended Price* of an order item:

```
:$.QPRICE = REALQUANT(:$.TQUANT)
* :$.PRICE * (100.0 - :$.PERCENT) / 100.0
* (:$$.CURRENCY = -1 ? :$.IEXCHANGE : 1.0);
```

Another example (this time of how to use the previous value of a specific column) is found in the CHECK-FIELD trigger for the **PARTNAME** column in the same form:

```
GOTO 1 WHERE :$1.PARTNAME = '' OR :$.ORDI = 0;
GOTO 1 WHERE :$.@ = :$1.PARTNAME;
```

Sometimes a trigger employing a dollar sign may refer to a non-existing form (e.g., $$ when there is no upper-level form). This sometimes happens when the trigger is included in more than one form. In such a case, ***Priority*** will consider the wildcard as representing the next form level down (the current form, to continue the above example).

Expressions and Column triggers often refer to the current form column. You can therefore use @ as a wildcard in place of this form column name (see example above). For instance, the link between sub-level and upper-level forms is generally made between columns having identical names. This linkage can be expressed using the @ wildcard.

The use of these wildcards makes it easier to read the trigger. They are also useful when employing an **#INCLUDE** command in a trigger (see below).

Note: The use of @ in a Row or Form trigger will stand for the name of that trigger (e.g., POST-FORM).

User-defined Variables

In addition to form-column variables, whose values are determined by the data in the form (or table) column, you may also define your own variables. For example, the following SQL statement employs a :CNT variable that counts the number of work orders opened for a given order item (see the CHECK-FIELD trigger for the **DUEDATE** column of the **ORDERITEMS** form):

```
:CNT = 0;
SELECT COUNT(*) INTO :CNT
FROM ORDSERIAL, SERIAL
WHERE ORDSERIAL.ORDI = :$.ORDI
AND ORDSERIAL.SERIAL = SERIAL.SERIAL
AND SERIAL.PEDATE > :$.DUEDATE
AND SERIAL.CLOSEDATE = 0;
```

When naming a variable, use the following rule of thumb:

- If the variable is included in a trigger in a standard form, use the appropriate four-letter prefix, so as to easily distinguish it from standard variables.

- If the variable is included in a trigger in your own form, the prefix is unnecessary.

For more details, see the customizing rules at the end of the chapter.

User-defined variables (such as :CNT) do not have an automatic starting value; rather, this must be set by a trigger. Consider, for instance, the following SQL statement in the first line of the POST-INSERT trigger for the **TRANSTRIG** form (this trigger is included in POST-INSERT triggers for the **TRANSORDER_D** form and many other forms like it), which sets the value of the :QUANT and :TQUANT variables on the basis of the quantity of shipped items:

```
SELECT 0 + :$.TQUANT,0 + :$.QUANT INTO :TQUANT,:QUANT
FROM DUMMY;
```

Note: Alternatively, you could use :TQUANT = :$.TQUANT and :QUANT = :$.QUANT, respectively.

Global Variables in Forms

Some forms are used to work with data that is relevant to more than one company, i.e., they refer to one of the companies defined in the **ENVIRONMENT** table or contain a trigger with a loop that runs for all companies (e.g., the *Itemized Rates* form). When defining your own variables for use in a multi-company form, you need to make sure to define them accordingly.

When a trigger is activated in a multi-company form, any user-defined variables included in the trigger automatically receive a prefix referring to the relevant company. For example, if a trigger that contains a variable called :SOMEVAR is activated from a record that is defined for the comp1 company, then while the trigger is running the variable will be renamed :_COMP1.SOMEVAR. If the same trigger is activated from a record that is defined for the comp2 company, then while the trigger is running the variable will be renamed :_COMP2.SOMEVAR.

In order to define a global variable for a multi-company form (i.e., a variable that receives the same value for all companies, regardless of the current record), add the prefix "GLOBAL." to the variable name (e.g., :GLOBAL.SOMEVAR). This prefix ensures that no company-specific prefix will be added when a trigger containing that variable is activated (and the variable can be used globally within the form).

Global variables are particularly useful when you want to define a loop that runs for all companies and updates records in each company (such as the loop in the *Itemized Rates* form). In such a case, you need to define a global variable that stores the name of the company from which the loop starts. If you use a local variable to store the company name, this variable will receive the prefix defined for the original company upon completing the loop – and will subsequently be empty.

The DUMMY Table

You will note that the SELECT statements illustrated above refer to the **DUMMY** table. This is a single-record, single-column table which may be included in an

SQL statement whenever values are assigned to a variable. While the SELECT statements in the above triggers could have referred to an ordinary database table, this would have caused the SQL mechanism to travel through all the table's records, which would take some time. It is therefore much faster to execute the SELECT via the **DUMMY** table. In fact, in ***Priority*** any SELECT ... FROM DUMMY statement does not even access the **DUMMY** table; hence, execution is even faster.

Text Form Variables

Using a form trigger, you can define a given text form as read-only when its upper-level form has a particular status, and you can prevent users from opening the text editor in non-HTML format. In the former case, set the :$.READONLY.T variable to 1; in the latter, set the :$.NOEDITOR.T variable to 1.

Example: See the PRE-FORM trigger in the **ORDERSTEXT** form.

By default, all text forms in ***Priority*** are HTML text forms. If you need to create a plain text form, set the :$.NOHTML.T variable to 1.

Example: See the PRE-FORM trigger in the **FTRIGTEXT** form.

Built-in Triggers

Essentially, the built-in triggers perform certain checks and, if the checks are successful, update the database accordingly. When a new record has been entered in a form, and an attempt is made to exit the line, the built-in triggers generate the following events:

- They check that values have been assigned to all the columns comprising any unique key(s).
- If the check succeeds, the line is successfully exited, the record is inserted into the form's base table, and an automatic value is given to the autounique key in that record, if there is one (increasing the counter by 1).
- If the check fails, an error message appears and the user cannot leave the line without clearing it or adding the missing values from the unique key.

When the user designates a unique key that already exists in the database, the built-in triggers ***automatically fill in all columns in the record***. This is one way to retrieve an existing record from the database. There is then an automatic shift from insert mode into update mode, and the retrieved record can be modified, as desired. If, however, no such record exists in the database, then you remain in insert mode and a new record is created.

When the user specifies a new record, the built-in triggers ***verify that imported data exist in their respective join tables***. If this verification check is successful, and all columns in any of the join table's unique keys have been filled in, then the built-in triggers ***automatically fill in any other columns in the form that are imported*** from that table.

When an existing record is updated in a form, the system performs the same verification checks that are activated during record insertion. Once all update checks are successful, the record is updated in the database.

Priority's built-in triggers also ***prevent the violation of referential integrity***. That is, they do not allow the deletion of any record containing a column that is imported into another form (including the column that links an upper-level form to its sub-level). Once all deletion checks are successful, the record is deleted from the database.

Finally, built-in triggers perform privilege checks. That is, they ***check whether or not the user is authorized to modify the database***. Only an authorized user will be able to insert, update or delete a record.

In sum, there are several types of built-in triggers:

- field triggers, which fill in and verify column values
- insert triggers, which verify the values in the row and insert the record in the table if the verification check is successful
- update triggers, which verify the updated values in the row and update the record if the verification check is successful
- delete triggers, which check for referential integrity and delete the record if the check is successful.

Field triggers are activated when the form column is exited. The other built-in triggers are activated when the row is exited.

The built-in insert, update and delete triggers only affect the form's base table. If the form is to insert, update or delete records of other tables, you must write your own POST-INSERT, POST-UPDATE and/or POST-DELETE triggers.

Example: The **STARTDATE** column in the *Service Calls* form (**DOCUMENTS_Q**) is from the **SERVCALLS** table (whereas the base table of the **DOCUMENTS_Q** form is **DOCUMENTS**). There are therefore POST-INSERT, POST-UPDATE and POST-DELETE triggers for this form that generate the insert/update/delete of records in the **SERVCALLS** table.

Field Triggers

If, when in insert mode, you fill in all the columns that make up the base table's unique key (which is often a single column, like **ORDNAME** or **CUSTNAME**), then the entire record is automatically retrieved and there is an automatic shift from insert mode to update mode. This function is carried out by ***Priority***'s built-in field triggers. The following code displays the built-in POST-FIELD trigger for the **ORDERS** form.

Note: Examples here and below are taken from a file created by the *Dump Form* utility (run via the *SQL Development* program, by selecting **Form** from the **Dump** menu). The result is the SQL queries of the form, including the triggers of that form, both built-in and user-defined.

```
ORDERS/ORDNAME/POST-FIELD TRIGGER:
----------------------------------
```

```
#line 1 ORDERS/ORDNAME/POST-FIELD
SET TRANSACTION;
GOTO 9 WHERE :NEXTPATTERNFLAG = 1;
SELECT CURDATE, BOOKNUM, FORECASTFLAG, DETAILS,
   REFERENCE, QPRICE, PERCENT, DISPRICE, VAT, TOTPRICE,
   QPROFIT, ADJPRICEFLAG, ADVBAL, ADVPERCENT,
   BSHN_OPENDATE, TYPE, CLOSED, PCLOSED, LEXCHANGE,
   DOER, AGENT, BRANCH, CURRENCY, CUST, DEAL,
   DESTCODE, LCURRENCY, ORD, ORD, ORD, ORD,
   ORDSTATUS, ORDTYPE, PAY, PHONE, PLIST, PROF, PROJ,
   SHIPTYPE, USER, WARHS
INTO :ORDERS.CURDATE, :ORDERS.BOOKNUM,
   :ORDERS.FORECASTFLAG, :ORDERS.DETAILS,
   :ORDERS.REFERENCE, :ORDERS.QPRICE,
   :ORDERS.PERCENT, :ORDERS.DISPRICE, :ORDERS.VAT,
   :ORDERS.TOTPRICE, :ORDERS.QPROFIT,
   :ORDERS.ADJPRICEFLAG, :ORDERS.ADVBAL,
   :ORDERS.ADVPERCENT, :ORDERS.BSHN_OPENDATE,
   :ORDERS.TYPE, :ORDERS.CLOSED, :ORDERS.PCLOSED,
   :ORDERS.LEXCHANGE, :ORDERS.DOER, :ORDERS.AGENT,
   :ORDERS.BRANCH, :ORDERS.CURRENCY, :ORDERS.CUST,
   :ORDERS.DEAL, :ORDERS.DESTCODE, :ORDERS.LCURRENCY,
   :ORDERS.LINKDOC, :ORDERS.NSCUST, :ORDERS.ORD,
   :ORDERS.ORDS, :ORDERS.ORDSTATUS, :ORDERS.ORDTYPE,
   :ORDERS.PAY, :ORDERS.PHONE, :ORDERS.PLIST,
   :ORDERS.PROF, :ORDERS.PROJ, :ORDERS.SHIPTYPE,
   :ORDERS.USER, :ORDERS.WARHS
FROM ORDERS
   WHERE ORDNAME = :ORDERS.ORDNAME;
LABEL 9;
:TABFORM = 22;#line 1 ORDERS/ORDNAME/POST-FIELD
   SELECT BRANCH INTO :$.BRANCH FROM USERSA WHERE
   :$.ORD = 0
   AND :$.BRANCH = 0 AND USER = SQL.USER;
COMMIT;
```

When inserting a new record, the built-in triggers verify that imported data exist in the join table:

Example: Built-in insert triggers do not allow you to specify an order made by a customer that does not appear in the **CUSTOMERS** table.

```
#line 1 ORDERS/CUSTNAME/CHECK-FIELD
GOTO 1 WHERE :ORDERS.CUSTNAME = '';
SELECT 'X'
```

```
FROM CUSTOMERS
   WHERE CUSTNAME = :ORDERS.CUSTNAME;

SELECT 192 INTO :SCREENMSG

FROM DUMMY WHERE :RETVAL = 0;

LABEL 1;
```

Once all columns in any of the join table's unique keys are filled in (provided that all verification checks are successful), ***Priority*** will automatically fill in any columns in the form that are imported from that table. Thus, once a valid customer number is designated in the *Sales Orders* form, the corresponding customer name will be filled in automatically.

```
#line 1 ORDERS/CUSTNAME/POST-FIELD

SET TRANSACTION;

SELECT CUSTDES, CUST, CUST, CURRENCY, LINKDATE, PAY,
   SHIPTYPE, MCUST, NSFLAG, PAYCUST, SECONDLANGTEXT,
   VATFLAG

INTO :ORDERS.CUSTDES, :ORDERS.CUST, :ORDERS.CUSTA,
   :ORDERS.CUSTCURRENCY, :ORDERS.CUSTLINKDATE,
   :ORDERS.CUSTPAY, :ORDERS.CUSTSHIPTYPE,
   :ORDERS.MCUST, :ORDERS.NSFLAG, :ORDERS.PAYCUST,
   :ORDERS.SECONDLANGTEXT, :ORDERS.VATFLAG

FROM CUSTOMERS
   WHERE CUSTNAME = :ORDERS.CUSTNAME;
```

Note: For technical reasons, the **CUSTDES** column is hidden in the **ORDERS** form, and the **CDES** column is displayed instead. A POST-FIELD trigger copies the value of **CUSTDES** into **CDES**.

Insert Triggers

When a line is exited, ***Priority***'s built-in insert triggers check that values have been assigned to all the columns comprising any unique key(s); they provide an automatic value to that table's autounique key (increasing the autounique counter by 1); and they insert the new record into the form's base table.

```
ORDERS INSERT TRIGGER:

SET TRANSACTION;

INSERT INTO ORDERS ( CURDATE, ORDNAME, BOOKNUM,
   FORECASTFLAG, DETAILS, REFERENCE, QPRICE, PERCENT,
   DISPRICE, VAT, TOTPRICE, QPROFIT, ADJPRICEFLAG,
   ADVBAL, ADVPERCENT, TYPE, CLOSED, PCLOSED,
   LEXCHANGE, DOER, AGENT, BRANCH, CURRENCY, CUST,
   DEAL, DESTCODE, LCURRENCY, ORD, ORD, ORD, ORD,
   ORDSTATUS, ORDTYPE, PAY, PHONE, PLIST, PROF, PROJ,
   SHIPTYPE, USER, WARHS)
```

```
VALUES ( :ORDERS.CURDATE, :ORDERS.ORDNAME,
   :ORDERS.BOOKNUM, :ORDERS.FORECASTFLAG,
   :ORDERS.DETAILS, :ORDERS.REFERENCE, :ORDERS.QPRICE,
   :ORDERS.PERCENT, :ORDERS.DISPRICE, :ORDERS.VAT,
   :ORDERS.TOTPRICE, :ORDERS.QPROFIT,
   :ORDERS.ADJPRICEFLAG, :ORDERS.ADVBAL,
   :ORDERS.ADVPERCENT, :ORDERS.TYPE, :ORDERS.CLOSED,
   :ORDERS.PCLOSED, :ORDERS.LEXCHANGE, :ORDERS.DOER,
   :ORDERS.AGENT, :ORDERS.BRANCH, :ORDERS.CURRENCY,
   :ORDERS.CUST, :ORDERS.DEAL, :ORDERS.DESTCODE,
   :ORDERS.LCURRENCY, :ORDERS.LINKDOC,
   :ORDERS.NSCUST, :ORDERS.ORD, :ORDERS.ORDS,
   :ORDERS.ORDSTATUS, :ORDERS.ORDTYPE, :ORDERS.PAY,
   :ORDERS.PHONE, :ORDERS.PLIST, :ORDERS.PROF,
   :ORDERS.PROJ, :ORDERS.SHIPTYPE, :ORDERS.USER,
   :ORDERS.WARHS);

SELECT 189 INTO :SCREENMSG FROM DUMMY WHERE :RETVAL
= 0;

SELECT ORD, ORD, ORD, ORD

INTO :ORDERS.LINKDOC, :ORDERS.NSCUST, :ORDERS.ORD,
   :ORDERS.ORDS

FROM ORDERS

WHERE ORDNAME = :ORDERS.ORDNAME;
```

Update Triggers

In addition to most of the functions performed by the insert triggers, ***Priority***'s built-in update triggers ensure that no column which links one form to another form has been updated.

Example: If the **PART** form were linked to the **PARTARC** form (which stores child parts) via the part catalogue number (**PARTNAME**), then that number could not be changed once a child part was assigned. This problem is easily resolved by linking the forms through the autounique key (**PART**), as the column in that key is not updateable. Rather, its value is automatically assigned by the system.

Delete Triggers

Priority's delete triggers prevent the violation of referential integrity. That is, they do not allow the deletion of any record containing a column that is imported into another form (including the column that links an upper-level form to its sub-level).

```
ORDERS/DELETE TRIGGER:

#line 1 ORDERS/DELETE

SELECT ENTMESSAGE('ORDERITEMS','F',0) INTO :PROGPARAM
FROM DUMMY;
```

```
SELECT 94 INTO :PROGMSG
FROM ORDERITEMS WHERE (:$1.ORD <> 0 AND ORD = :$1.ORD );
```

Creating Your Own Triggers

Triggers are specified in the *Form Column Triggers* form and its sub-level, *Form Column Triggers - Text*, when they are to be activated by movement out of a given column. They are specified in the *Row & Form Triggers* form and its sub level, *Row & Form Triggers - Text*, when they are to be activated by exiting the line or the form.

Notes:

Once a trigger has been formulated for any given form column, a check mark appears in the *Triggers* column of the *Form Columns* form. This flag makes it easy to locate those columns for which user-designed triggers have been created.

You can track changes to custom triggers once they have been included in prepared version revisions. For details, see Chapter 9.

It is easier to use ***Priority***'s text editor to write trigger text rather than using the regular form. However, this must be done with caution. Specifically, it is imperative to make sure the entire trigger appears in the form before moving to the text editor; otherwise, any undisplayed lines will be lost! So, if you have cleared any lines in the form or retrieved only some of the records, be sure to retrieve all records before you run the text editor.

►To access the text editor, press **F6**.

Tip: You can replace the built-in text editor with any editor of your choosing (e.g., Notepad). To do so, run the *Define External Editor* program (*System Management → Generators → Procedures*), indicating the full path to the file you want to use (e.g., Tabula External Editor= C:\Windows\Notepad.exe) in the input screen. Alternatively, you can open the *tabula.ini* file (in *C:\Windows*) and, under the {Environment} section, revise the line for Tabula External Editor, giving the file path.

Types of Triggers

The following types of triggers may be created:

CHECK-FIELD	Performs verification checks on a value specified for a form column.
POST-FIELD	Performs operations once form column check is successful.
PRE-INSERT	Performs verification checks before a record is inserted into the database.
POST-INSERT	Performs operations once a record is successfully inserted.

PRE-UPDATE	Performs verification checks before a record is updated in the database.
POST-UPDATE	Performs operations once a record is successfully updated.
PRE-DELETE	Performs verification checks before a record is deleted from the database.
POST-DELETE	Performs operations once a record is successfully deleted.
PRE-FORM	Performs operations before a form is opened.
POST-FORM	Performs operations when a form is exited, provided there were insertions, updates or deletes in the form.
CHOOSE-FIELD	Creates a list of values from which the user can choose when filling in a specific field (for short lists).
SEARCH-FIELD	Creates a list through which the user can search for the needed value of a given field (for long lists).

Except for PRE-FORM triggers, which are ***always*** activated, triggers ***will not be activated*** unless the user has made an addition or change in the column, row or form. Furthermore, CHOOSE-FIELD and SEARCH-FIELD triggers are only activated when the user accesses a Choose list or Search list, respectively.

Order of Trigger Execution

Before demonstrating the usage of each of the user-designed triggers, a brief explanation of their sequence of execution is in order:

- CHECK-FIELD triggers precede POST-FIELD triggers.
- Built-in CHECK-FIELDs precede user-designed CHECK-FIELDS.
- Built-in POST-FIELDs precede user-designed POST-FIELDS.
- PRE- triggers precede their respective POST- triggers.
- Built-in triggers are executed after PRE- triggers, before POST- triggers.
- Standard and custom triggers are sorted alphabetically, so you should name your own triggers accordingly. For example, to run your own trigger after a standard POST-INSERT trigger, use POST-INSERT_AXXX or ZXXX_POST-INSERT (where XXX is part of the prefix you normally use for this customer).

Trigger execution is discontinued when an END or ERRMSG command succeeds. It is not discontinued if a WRNMSG command succeeds.

If a CHECK-FIELD trigger is discontinued, then the corresponding POST-FIELD triggers (built-in and user-designed) will not be activated. Similarly, if a verification check fails in any of the PRE- triggers, then activation of that trigger is discontinued, and the corresponding built-in and POST- trigger will not be activated.

Example: Suppose the **PARTNAME** column of the **ORDERITEMS** form has been assigned both a CHECK-FIELD and a POST-FIELD trigger. The former ensures that the part number is not updated once it has been successfully inserted in a record (rather, the user must delete the record for the unwanted part number and insert a new record). The POST-FIELD trigger generates a unit price for the ordered part based on the customer's price list. If the CHECK-FIELD should fail

(i.e., the user has attempted to update the part number), then the price of the part will not be generated. Nor will the built-in POST-FIELD trigger (which fills in the internal number and part description on the basis of the designated part number) be activated.

Naming Customized Triggers

Customized triggers should be given special names (which must be added to the *List of Triggers* form; this form can be accessed by pressing **F6** from the *Row & Form Triggers* form or from the *Form Column Triggers* form). These names have to include key strings which indicate the type of trigger involved, together with additional characters that refer to the customization. For instance, customized CHECK-FIELD triggers must include the strings "CHECK" and "FIELD" (in capital letters). Thus, a customized CHECK-FIELD for an installation at CRR Holding Company might be labelled CRRH_CHECK-FIELD.

Customized trigger names must follow a number of rules:

- They can only contain alphanumeric values, the underline sign (_) and the hyphen (–).
- They must begin with a letter.
- They must not include spaces.
- They must include a four-letter prefix or suffix, similar to what you use throughout the system for this customer. The difference is that you will need to choose the appropriate first letter of the prefix (or suffix) for sorting purposes (see example above).
- They must include the required key strings that identify trigger, separated by hyphens (i.e., PRE-, POST-, -FIELD, -FORM, -INSERT, -UPDATE, -DELETE, CHECK-, CHOOSE-).

Notes:

Key strings need not be consecutive (e.g., they may be separated by additional strings).

You can combine key strings to create, for instance, a customized trigger that operates both as a POST-INSERT and a POST-UPDATE (e.g., ARRH_POST-INSERT-UPDATE).

SEARCH-FIELD triggers are the exception to the rule. You cannot create a customized SEARCH-FIELD, but rather must use the standard trigger. For details, see the customizing rules at the end of this chapter.

Creating Column Triggers

CHECK-FIELD

CHECK-FIELD triggers perform verification checks on the value specified for a column. Note that the value must have been inserted or updated by the user. The check will not be performed if the user simply moves the cursor through this column, without making any changes in it (even if the value was filled in by a trigger after record retrieval).

CHECK-FIELD triggers should be used when you wish to restrict the values that may be specified (in addition to the validation checks built into the system). The error/warning message is specified in the sub-level form (*Error and Warning Messages*). It should be assigned a number greater than 500.

Example 1: The CHECK-FIELD trigger for the **TYPE** column of the **PART** form is:

```
ERRMSG 4 WHERE :$.TYPE NOT IN ('O','R','P');
```

This trigger restricts the part type to *O*,*R,* or *P*. If any other value is specified, the user will not be able to exit the column, and an error message will be generated: “Specify P (part), R (raw material) or O (other).”

Example 2: The CHECK-FIELD trigger for the **TQUANT** column of the **ORDERITEMS** form warns the user if the quantity specified in the current column of the current form is less than zero (“The designated quantity is a negative number!”):

```
WRNMSG 105 WHERE :$.@ < 0;
```

POST-FIELD

POST-FIELD triggers cause operations to be performed once the value specified for the column has successfully passed the verification checks. They are particularly useful for filling in values.

Example: The POST-FIELD trigger for the **SUPNAME** column of the **PORDERS** form inserts the current date (**SQL.DATE8**, a system function) into the **CURDATE** column when opening a new purchase order if there is currently no date in that column.

Note: When a POST-FIELD trigger changes a value of another form column, the POST-FIELD of that other column (if there is one) will also be activated, but its CHECK-FIELD will not.

CHOOSE-FIELD

CHOOSE-FIELD triggers create a short list of values from which the user can choose. Each column in the CHOOSE-FIELD query is restricted to 64 characters.

Example: The CHOOSE-FIELD trigger for the **PARTNAME** column in the *Purchase Order Items* form (**PORDERITEMS**) provides a list of the vendor’s parts (:$$.SUP):

```
SELECT DISTINCT PARTDES,PARTNAME
FROM PART WHERE PART =
(SELECT PART FROM SUPPART WHERE SUP = :$$.SUP AND VALIDFLAG = 'Y')
AND PART <> 0
ORDER BY 1;
```

The above type of CHOOSE-FIELD is a regular SQL query where both arguments must be of **CHAR** type. In order to display a number, you must first convert it into a string using the **ITOA** function (see Chapter 2).

The first argument in the query is the description and the second is the value to be inserted in the column where the Choose list is activated. You can also add a third argument for sorting purposes. If you want to display a single value in the Choose list, you must add the empty string to the CHOOSE-FIELD query.

Example: See the CHOOSE-FIELD trigger for the **CPROFNUM** column in the **ORDERS** form. Its first argument displays the quote's **DETAILS** and **PDATE** columns; the second is the quote number (the value that will be filled in); the third determines the sort, which is by the quote's expiration date.

If the third argument (the one you wish to sort by) is a numeric value, using the ITOA function alone will not result in correct sorting — for example, 10 would precede 5. To avoid this, you can use the function ITOA(*m,4*), which will result in 1 becoming 0001, 5 becoming 0005 and 10 becoming 0010, so that the order when sorted will be correct.

The first argument in the query is stored in the :PAR4 system variable, where it can be used by other triggers assigned to the same column.

Example: See the BUF2 trigger for the **SERNUM** column in the **DOCUMENTS_Q** form (called from the CHECK-FIELD trigger for that form column). It uses the :PAR4 variable, containing the part description that corresponds with the chosen serialized part, to determine the corresponding part number.

A CHOOSE-FIELD trigger can also display a list of constant values, taken from form messages. When using this type of CHOOSE-FIELD, the messages must be structured as follows: *Value*, *Description*.

Example: The CHOOSE-FIELD trigger for the **TYPE** column of the **LOGPART** form is:
MESSAGE BETWEEN 100 AND 102;
This trigger will display form messages 100, 101 and 102. Message 100 (for example) is then defined as: P, "Make" item.

You can also create a Choose list from a set of queries. Results are then combined into a single list, creating a Union Choose. Alternately, you can define the SELECT statement so that once one of the queries succeeds, no subsequent queries are run. To do so, use the syntax: SELECT /* AND STOP */ …

Example: Use a Union Choose in the **ORDERS** form to create a Choose list comprised of the customer's part list and the representative customer's part list. For an example of SELECT /* AND STOP */, see the CHOOSE-FIELD trigger for the **PARTNAME** column in the **ORDERS** form.

Note: The sort in a Union Choose is determined by the first column that is retrieved. That is, the sort defined for each query, as well as the order of the queries, is ignored. If you want to override this default, using instead the sequence of results retrieved from each query, add the following comment anywhere in the trigger (note the space after the first asterisk and before the second one):
/* NO SORT */

For an example of such usage, see the CHOOSE-FIELD trigger for the **ACTNAME** column in the **ALINE** form.

Finally, you can use a variation of the trigger, called MCHOOSE-FIELD, to create a Multiple Choose. This allows the user to select more than one value from the Choose list at a time.

Example: See the MCHOOSE-FIELD trigger in the **ORDNAME** column of the **DOCORD** form.

Notes:

If there are no values in the Choose list (e.g., no parts were defined for the vendor), or if it contains more records than the number specified in the CHOOSEROWS system constant, a SEARCH-FIELD trigger (if defined) is activated instead.

A CHOOSE-FIELD trigger can also be defined for an entire form (see below).

Creating Row Triggers

PRE-INSERT

PRE-INSERT triggers perform operations on a new record before it is inserted in the database (generally verification checks that are not performed by the built-in Insert triggers). This type of trigger is useful, for example, for checking the contents of several form columns before the line is exited.

Example: The PRE-INSERT trigger for the **CASH_CASH** form (*Cashiers*) verifies that a GL account is attached to the cashier:

```
WRNMSG 1 WHERE :$.ACCOUNT = 0;
```

POST-INSERT

POST-INSERT triggers cause the performance of certain operations once the record has been successfully inserted in the database.

Example: The POST-INSERT trigger in **DOCUMENTS_Q** inserts a record into the **SERVCALLS** table.

PRE-UPDATE

PRE-UPDATE triggers perform verification checks before a record is updated in the database. Generally, they are similar to PRE-INSERT triggers.

Example: The PRE-UPDATE trigger in the **CASH_CASH** form verifies that a GL account is attached to the cashier:

```
WRNMSG 1 WHERE :$.ACCOUNT = 0;
```

POST-UPDATE

POST-UPDATE triggers cause the performance of operations once the record has been successfully updated.

Example: The POST-UPDATE trigger in the **DOCUMENTS_Q** form updates those form columns that come from the **SERVCALLS** table.

PRE-DELETE

PRE-DELETE triggers perform verification checks before a record is deleted from the database (in addition to the checks that are built into the system).

Example: The PRE-DELETE trigger in the **ORDERITEMS** form warns the user that tries to delete a line containing a bonus item:

```
WRNMSG 334 WHERE :$.BONUSFLAG IN ('Y','F');
```

POST-DELETE

POST-DELETE triggers cause the performance of operations once a record has been successfully deleted.

Example: The POST-DELETE trigger in the **ORDERITEMS** form deletes the relevant record from the **ORDERITEMSA** table:

```
DELETE FROM ORDERITEMSA WHERE ORDI = :$.ORDI;
```

CHOOSE-FIELD

In addition to creating a CHOOSE-FIELD for a specific form column, you can also define a CHOOSE trigger at the form level. This ensures that, whenever the designated table column appears in another form, the Choose list will be defined by the current form. In fact, in order to override it and create a different Choose list, you will have to write a new CHOOSE-FIELD trigger for the form column in question.

Example: The CHOOSE-FIELD trigger in the **ORDSTATUS** form causes the Choose list of order statuses to be taken from this form. That is, any form that includes the **ORDSTATUSDES** column from the **ORDSTATUS** table takes its CHOOSE-FIELD from the **ORDSTATUS** form.

SEARCH-FIELD

SEARCH-FIELD triggers create a long Search list which the user can use to search for a desired value (e.g., to select a customer out of a list of thousands of customers). There are two main types:

- the SEARCH-NAME-FIELD, which searches by number
- the SEARCH-DES-FIELD, which searches by name or description.

Example: The SEARCH-NAME-FIELD trigger in the **CUSTOMERS** form performs the search on the customer number, whereas the SEARCH-DES-FIELD trigger performs it on the customer name.

You can also create a Multiple Search, that is, a trigger that allows the user to select more than one value from the Search list at a time (similar to MCHOOSE-FIELD). To do so, include the following comment in the trigger:
/* MULTI */.

Notes:

A third type (SEARCH-EDES-FIELD) is used only in a non-English system, to search by the name/description in English.

SEARCH-FIELD triggers are the exception to the rule, as you cannot create a customized trigger of this type. Instead, you must use standard SEARCH-FIELD triggers. For details, see the customizing rules at the end of this chapter.

SEARCH-FIELD triggers can only display table columns with a width of up to 59 characters.

If the user selects more than one value in a Multiple Search and the form generates a warning or error message, insertion of the selected values will stop.

Creating Form Triggers

PRE-FORM

PRE-FORM triggers perform operations before the form is opened. This applies to all root forms, as well as sub-level forms for which the *Automatic Display* column of the *Sub-Level Forms* form (a sub-level of the *Form Generator)* is blank. This type of trigger may be used, for example:

- to reset the value of a user-defined variable
- to generate a warning or error message concerning retrieved data
- to retrieve and display all records when the user opens the form — :KEYSTROKES = '*{Exit}';
- to refresh all retrieved records in a form following a Direct Activation — :ACTIVATEQUERY = 1;
- to deactivate data privileges in a form — :$.NOCLMNPRIV.T = 1;
- to deactivate data privileges for a specific table in a form: in a new PRE-FORM trigger for the form in question, define the :$.NOTBLPRIV.T variable with the name of the desired table; if the table you want to exclude has a join ID, this should also be specified.

Example: In order to exclude the **AGENTS** table from data privilege restrictions defined for the **ORDERS** form, add the following line:
:$.NOTBLPRIV.T = 'AGENTS'

Or, if there is a join ID of 5:
:$.NOTBLPRIV.T = 'AGENTS.5'

Tip: To activate a PRE-FORM trigger after every query, include the line :PREFORMQUERY = 1;

POST-FORM

POST-FORM triggers perform operations upon exiting the form (provided that the user has made at least one change in the database). This type of trigger is useful, for example, for updating values in the upper-level form on the basis of values specified in the sub-level form.

Example: The POST-FORM trigger in the **TRANSORDER_K** form includes the following lines (among others):

```
UPDATE SERIAL SET KITFLAG = 'Y' WHERE SERIAL = :$$.SERIAL
AND EXISTS
(SELECT 'X' FROM KITITEMS WHERE SERIAL = :$$.DOC AND TBALANCE > 0
AND KITFLAG = 'Y');
GOTO 1 WHERE :RETVAL > 0;
```

This part of the trigger updates the *Missing Components* column in the *Work Orders* form upon exiting the *Issues to Kits* sub-level form.

Error and Warning Messages

Triggers which perform verification checks should include error message (ERRMSG) or warning message (WRNMSG) commands. When they succeed, both these commands generate a message. However, whereas trigger execution continues when a warning message command is successful, it halts once an error message command is successful.

Tip: The system manager can change any warning into an error message via the *Privilege Explorer*. For details, run the *Privilege Explorer Wizard*.

Activating the Command

To activate an ERRMSG or WRNMSG command, use the following syntax:

ERRMSG *number* [**WHERE** *condition*];

WRNMSG *number* [**WHERE** *condition*];

The number refers to the message that will be generated. For example,

ERRMSG 4 WHERE :$.TYPE NOT IN ('O','R','P');

will cause message 4 (“Specify P (part), R (raw material) or O (other).”) to appear if the part type is not *O*, *R* or *P*.

Specifying the Message Content

For any given form, all ERRMSG and WRNMSG commands appearing in any trigger must be accompanied by error and warning messages. Moreover, it is important to ensure that the message is assigned the same number as the one referred to in the appropriate command. If the form preparation mechanism encounters a trigger with an ERRMSG or WRNMSG command and no message to accompany it, a warning will appear in the *Warnings Report*:

There is no message number X for Y form (appears in Z trigger).

If this warning is not heeded, and the form is loaded, the trigger will work normally. However, ***instead of*** its designated error or warning message, the user will receive the above message.
The content of each message is generally written in the *Error & Warning Messages* form. Each new message should be assigned a number greater than 500. Actually, there are three *Error & Warning Messages* forms that can be used interchangeably:

- the **FORMMSG** form, a sub-level of the *Form Generator* form;
- the **TRIGMSG** form, a sub-level of the *Row & Form Triggers - Text* form, which allows you to view the contents of the Row or Form trigger that generates the message;
- the **TRIGCLMSG** form, a sub-level of the *Form Column Triggers - Text* form, which allows you to view the contents of the Column trigger that generates the message.

If the message is longer than a single line, it may be continued in the sub-level of each of the above forms.

Storing the error/warning messages in separate tables from the triggers themselves gives them autonomous status. This is useful in two respects. First, it enables the messages to be displayed in the *Form Messages* dictionary. Hence, messages for different forms can be retrieved together, which helps the form designer to unify their style. Second, it enables ***Priority***'s translation facility to take the messages into account. Thus, while the contents of triggers will remain the same, the error and warning messages that are generated by these triggers can be displayed in another language.

In any given error or warning message, you can refer to a specific ***Priority*** entity using a special format: **{*entity_name*.**{ **F** | **R** | **P** | **M** } **}**, where F = form, R = report, P = procedure, M = menu. That is, you designate the entity name and type, and the entity's title will appear in their place. This format is useful because entity names are rarely changed, whereas titles are rather likely to be modified in upgraded or customized versions. In this way, the most up-to-date title will appear in your message.

Example: You can create a warning message for a trigger in the **SHIPTO** (*Shipping Address*) form which refers to the *Customers* form:
The shipping address is identical to the customer's mailing address. See the {CUSTOMERS.F} form.

Remember to check that the entity name and type have been correctly written, that is, the entity you specified really exists.

Tip: Run the *Check Entity Names in Msgs/Help* program (*System Management → Dictionaries → Messages and Help Text*).

You can also create a warning message (WRNMSG), error message (ERRMSG) or send mail message (MAILMSG) that displays the content of a text file. Within the trigger in question, define a variable of **FILE** type called MESSAGEFILE and specify **msg_number** = 1000000. This message number should *not* appear in the *Error & Warning Messages* form.

Example: A CHECK-FIELD trigger might contain the following code:
```
SELECT SQL.TMPFILE INTO :MESSAGEFILE FROM DUMMY;
SELECT 'Sample message' FROM DUMMY UNICODE :MESSAGEFILE;
ERRMSG 1000000;
```

Message Parameters

An error or warning message can include parameters (a maximum of three per message) — <P1>, <P2> and <P3>. The values to be assigned to these parameters are defined in the trigger that generates the message, by means of the system variables :PAR1, :PAR2 and :PAR3.

Example: The CHECK-FIELD trigger for the **PARTNAME** column of the **PORDERITEMS** form checks that the specified order item is sold by the vendor to which the order is made:
```
WRNMSG 140 WHERE NOT EXISTS /* Don't give warning for nonstandard*/
(SELECT 'X' FROM PARTPARAM WHERE PART =
(SELECT PART FROM PART WHERE PARTNAME = :$.@)
AND NSFLAG = 'Y')
AND NOT EXISTS
(SELECT 'X' FROM SUPPART, PART WHERE SUPPART.SUP = :$$.SUP
AND SUPPART.VALIDFLAG = 'Y'
AND SUPPART.PART = PART.PART AND PART.PARTNAME = :$.@);
```
Warning message 140 would then be: Vendor <P1> does not supply this part. PAR1 is filled in by the appropriate vendor number.

The variables :PAR1,:PAR2 and :PAR3 are of **CHAR** type. If you wish to assign a form column variable which is of a different type to a message parameter, you will have to first convert it to a string (use **ITOA** for an integer and **DTOA** for a date; see Chapter 2 for details).

Example: To insert the order date into a message parameter, include the following statement in the trigger:
```
:PAR1 = DTOA (:$.CURDATE, 'MM/DD/YY')
```

Sending a Mail Message

Mail messages are similar to error and warning messages, except that they are sent by internal or external mail to designated recipients and can be accompanied by an attachment. Thus, you need to define one or more users and/or e-mail addresses to which the message is to be sent, as well as the filename (if you are sending an attachment).

Tip: Use ***Priority*** groups to define multiple users and/or e-mail addresses.

Examples:

(1)
```
MAILMSG 9 TO USER :NEXTSIGN WHERE :NEXTSIGN <> 0 AND :NEXTSIGN <>
SQL.USER;
```
(2)
```
:EMAIL = 'yuval@priority-software.com';
:FILE = 'C:\tmp\msg.doc';
MAILMSG 5 TO EMAIL :EMAIL DATA :FILE;
```

Controlling the Appearance of Line Breaks within a Message

Note: This feature does not support output of bi-directional languages such as Hebrew and Arabic.

In certain cases, the message you want to send will be longer than a single line (i.e., message text continues into the *Error/Warning Message (cont.)* sub-level form). In such cases, the message text will be broken up into multiple lines.

In order to control the appearance of line breaks within a message, include the following string in the message:

<!--| priority:priform |-->

Priority will treat any text that follows this string as HTML code. This enables you to include HTML tags such as line breaks in your message, affording you a greater degree of control over the appearance of text in the message.

Example: The following e-mail is sent using the command:
MAILMSG 605 TO USER :USER;
Where warning message 605 contains the following:
Service Call request number: <P1> was updated by Customer.
and the *Error/Warning Message (cont.)* sub-level form contains the following:
<P2><P3>

The resulting message is sent without using HTML tags:
Service Call request number: ILSC123456 was
updated by Customer.
Customer: CRR
Holding Company
Date&Time: 18/10/08 14:28

The second e-mail is sent using the identical command:
MAILMSG 605 TO USER :USER;
Where warning message 605 contains the string:
<!--| priority:priform |-->
And the *Error/Warning Message (cont.)* sub-level form contains the following:
Service Call request number: <P1> was updated by Customer.
<P2>
<P3>

After line breaks are inserted using HTML tags, the same message appears as follows:
Service Call request number: ILSC123456 was updated by Customer.
Customer: CRR Holding Company
Date&Time: 18/10/08 14:28

Updating the History of Statuses Using MAILMSG

The MAILMSG command can also be used to update the *History of Statuses* (**DOCTODOLISTLOG**) form, by including the following syntax in the appropriate POST- trigger:

:PAR1 = *statustype*; /* the type of document whose assigned user and status you want to record in the **DOCTODOLISTLOG** form */
:PAR2 = *:iv*; /* the autounique value of the record whose assigned user and status you want to record in the **DOCTODOLISTLOG** form */
MAILMSG 1 TO USER -2;

The parameters :PAR1 and :PAR2 are used to indicate a unique form and record from which to retrieve the status and assigned user. The MAILMSG command generates a new line in the *History of Statuses* form with the user and status currently assigned to the specified record.

Example: To add a new line to the *History of Statuses* sub-level of the *Sales Orders* (**ORDERS**) form for Order No. A00098, use the following syntax:
:PAR1 = 'O'; /* the *Type* defined for all records in the **ORDERS** form */
:PAR2 = '15982'; /* the internal *Document (ID)* of Order Number A00098 */
MAILMSG 1 TO USER -2;

You can also add this syntax to a custom POST- trigger in order to record other changes to a form record.

Examples:
(1) To update the status history for the *Tasks* form after a change in the *Notes* (**CUSTNOTESTEXT**) form, add the above syntax to a new custom POST-FORM trigger.
(2) To do the same after a change in the *Start Date* of the task, add the above syntax to a new custom POST-UPDATE trigger.

Note: This feature can only be used in a sub-level form, not in the root form. For example, you cannot update the status history of an order from the *Sales Orders* (**ORDERS**) form.

Sending a Link to a Document using MAILMSG

You can also use the message parameters to include a link to a document in a message.

Example: Message 1 is defined as: "Here is a link to order <P1.ORDERS.F>."
:PAR1 = 'SO1212888';
MAILMSG 1 TO USER :USER;

Changing Column Titles Dynamically

Column titles can be set dynamically, so that they are assigned when the user enters the form. This option is available for forms that are not screen-painted and which have a value of *T* in the *One-to-many* column of the *Form Generator*.

To change the column title, use a variable made up of the form name, the title name and "TITLE".

Example: To change the title of the "TEST" column in the "MY_FORM" form to "New Title," write in the PRE-FORM trigger:
MYFORM.TEST.TITLE = 'New Title';
See also the **FRGROUPS_DET** form.

Including One Trigger in Another

The #INCLUDE command enables you to use the same trigger more than once without rewriting it.

To include the same trigger elsewhere, use the following syntax (note that there is no semi-colon at the end of the statement):

#INCLUDE *form_name* / *trigger_name*

#INCLUDE *form_name* / *form_column_name* / *trigger_name*

When a trigger is included in one or more other triggers, the entire contents of the former are inherited by the latter. Thus, if the CHECK-FIELD trigger for the **TYPE** column in the **PART** form is identical to the CHECK-FIELD trigger for the **TYPE** column in the **LOGPART** form, you could write the trigger for one column and include it in the other:

CHECK-FIELD *for* **TYPE** *column in* **PART** *form:*
ERRMSG 4 WHERE :$.TYPE NOT IN ('O','R','P');

CHECK-FIELD *for* **TYPE** *column in* **LOGPART** *form:*
#INCLUDE PART/TYPE/CHECK-FIELD

Moreover, you can write more statements to the latter (*including*) trigger which do not apply to the former (*included*) trigger (before or after the include command).

In contrast, any additions to the included trigger will automatically be attached to the including trigger as well. That is, if changes are made in an included trigger, this will affect all forms that include it; thus, all forms involved will have to be prepared as executable files by the form preparation mechanism.

If the included trigger is deleted, the error will be revealed during the form preparation of any forms that include that trigger.

Tip: To view the original trigger, move to the #INCLUDE line and press **F6**. The trigger text appears in the sub-level form, *Row & Form Triggers – Text*.

Using Buffers

Often, it is more efficient to include only a portion of a trigger. That is, several triggers may share the same set of SQL statements, but each trigger also has additional statements of its own. In such a case, a special type of trigger, called a “buffer,” should be defined. This buffer should hold all the shared SQL statements and it should be included in any trigger that uses this group of statements.

Example: Almost no document in the system (e.g., *Goods Receiving Voucher, Customer Shipments, Customer Returns*) can be revised once it's final. The relevant sub-level forms (*Received Items, Shipped Items, Returned Items*) should therefore include the same check. Thus, the PRE-INSERT trigger of all these sub-level forms includes the same trigger that performs the check:
#INCLUDE TRANSTRIG/BUF10 /* Check CANCEL and FINAL */

Naming Buffers

A buffer may either be numbered (BUF1, BUF2, BUF3, ..., BUF19) or assigned a trigger name that hints at its usage. Before any buffer name can be used, it must first be added to the *List of Triggers form*, which can be accessed by pressing **F6** from the *Row & Form Triggers* form or from the *Form Column Triggers* form. Numbered buffers already appear in this list.

The restrictions on buffer names are virtually identical to those of customized triggers (see above). The only difference is that, of course, no key strings may be used.

Nesting INCLUDE Commands

The #INCLUDE command can be nested. That is, one trigger can include another, which in turn includes a third trigger (and so on).

Example: The PRE-INSERT trigger of the **TRANSTRIG** form includes many buffers from the same form. (The **TRANSTRIG** form is a special form that only contains triggers included in other forms.)

Advantages of the Wildcards

Wildcards ($ and @) are very useful when including one trigger in another. This is because the wildcard has a relative meaning, which depends upon the form in which the trigger is activated.

Example: :$$.DOC in the PRE-INSERT trigger of the *Received Items* form refers to :DOCUMENTS_P.DOC, whereas in the PRE-INSERT trigger of the *Shipped Items* form, it refers to :DOCUMENTS_D.DOC.

Error and Warning Messages

Triggers inherit not only all SQL statements from the included trigger (or buffer), but also their accompanying error and warning messages. The scope of the messages is all the triggers written for this form.

Checking Trigger Usage

Caution must be exercised when revising a trigger that is included in other triggers, as any changes in the former will obviously affect the latter. You can view any triggers that include the current trigger in the *Use of Trigger* sub-level form of *Form Column Triggers* and of *Row & Form Triggers*.

Trigger Errors and Warnings

Form preparation will fail if any major trigger errors are encountered. The main trigger errors are:

- SQL syntax errors
- illegal or irresolvable variable types
- syntax errors in #INCLUDE commands
- #INCLUDE commands that refer to non-existent forms, form columns or triggers
- form column variables that refer to non-existent form/form column combinations (e.g., :ORDERS.CUSTNAME, when there is no **CUSTNAME** column in the **ORDERS** form)
- ERRMSG or WRNMSG commands that refer to message numbers not specified for the form in question.

In addition, warning messages (without causing form preparation to fail) might be generated when the same local variable is used for two distinct types. Even though form preparation succeeds in this case, it is nevertheless recommended that you take care of all the problems that generated the warning messages.

Example: The same variable name contains a value of **INT** type in one trigger and a value of **CHAR** type in another.

Form Preparation

A form which has been constructed or revised cannot be accessed on screen until it is prepared as an executable file. In most cases, form preparation is activated automatically when you attempt to open an unprepared form. You can also activate it manually:

- Run the *Form Preparation* (**FORMPREP**) program (*System Management→ Generators→ Forms→ Form Preparation*), which can prepare multiple forms simultaneously.
- Run the *Prepare Form* program by Direct Activation from the *Form Generator* for a specific form.

Form preparation will fail if errors are encountered (and you will be referred to an *Errors Report*, located in the same menu). There are several types of major errors: trigger errors, problems with form content and problems with form display.

Examples of possible problems:

- A column in the base table's autounique or unique key is missing from the form
- A calculated column has been assigned a width that is unsuitable for its column type (e.g., a width of 11 for a **DATE** column)
- A form column is neither derived from a table column nor defined by an expression.

Note: In rare cases, the *Form Preparation* program fails to replace the older version of the form. If that occurs, retrieve the form in question in the *Form Generator* and run the *Reprepare Form* program by Direct Activation.

Loading a Form

The *Load Form* program (*System Management → Generators → Forms*) allows you to access a form on screen without going through any menus. A similar program (*Open Form*) can be run by Direct Activation from the *Form Generator.* These interchangeable programs are particularly useful during the form's development stages.

Note: An unprepared form cannot be opened in this manner; it must first be prepared.

When loading a form, ***the designated form may not be linked to any upper-level form.*** To access a sub-level form, load the most upper-level form in its tree (the root form) and use that form to access sub-level forms.

Example: To access the **ORDERITEMS** form, load the upper-level **ORDERS** form and then access the sub-level form via a record in the *Sales Orders* form.

Help Messages

You can use the *Form Generator* to create on-line help messages for an entire form or specific form columns. Form help is specified in the *Help Text* form, a sub-level of the *Form Generator* form; column help is designated in the *Help Text* form, a sub-level of the *Form Columns* form. Whenever help messages (for the entire form or for any column) are added or modified, the date and time of this revision appear in the *Help Date* column of the *Form Generator* form.

Line Breaking

Line breaking for help messages is set automatically. Therefore, you can freely edit the text in the *Help Text* form without having to worry about the length of lines. You should therefore refrain from the use of hyphenation. To force the beginning of a line, use the code \n.

Example: The help for the **CPROFSTATS** form says:
Use this form to view statuses of price quotations and their attributes. \n
\n
To define statuses, along with their attributes and rules, run {VISCPROFSTATS.P} from the list of Direct Activations.

Referring to Other Entities

Often, you will want to refer to other ***Priority*** entities in your help text. As entity titles are easily changed (even by the user), whereas their names are relatively fixed by the time you begin to create help messages, ***Priority*** offers

a mechanism for referring to an entity by *name* in the help text; its current *title* will consequently be displayed when the help is called up by a user.

To refer to a specific entity in a help message, designate the entity name followed by a period and its type (as illustrated in the above example), where *F*=form, *R*=report and *P*=procedure, enclosing all of this in curved brackets.

When your help messages are more or less complete, you should check that all designated entity names and types (e.g., {VISCPROFSTATS.P}) are correct. To do so, run the *Check Entity Names in Msgs/Help* report (*System Management → Dictionaries → Messages and Help Text*). Any mistakes that are uncovered will have to be corrected in the appropriate *Help Text* form.

Repreparing the Form

Any modifications of help messages, including the creation of new ones, will not be seen by the user until the form in question has been reprepared. Unlike other changes to the form, modifications of help texts require you to remove the existing form preparation and execute it again. To do so, run the *Reprepare Form* program by Direct Activation from the *Form Generator*.

Rules for Customizing Forms

- ***Never*** use a standard base table to create your own form. Create your own table instead.
- Include the appropriate four-letter prefix (e.g., **XXXX_ORDERS**) in the name of any form you create.
- When modifying a standard form, any newly added column must start with your four-letter prefix.
- You cannot delete a standard column from a standard form.
- When creating your own multiple joins, use a join ID and column ID greater than 5.
- Any trigger you add must start with a four-letter prefix or end with a four-letter suffix. Choose the first letter in the prefix/suffix for sorting purposes; the rest of the prefix/suffix should be the one normally used for this customer.
  Important Note: SEARCH-FIELD triggers are the one exception to this rule, as their name cannot be changed. Instead, you have to use the standard trigger. This creates the slight risk that your trigger will be overwritten by a standard SEARCH-FIELD trigger that is changed in future software revisions.
- Any variable that you add to a standard form should start with the same four-letter prefix. If it does not, you run the risk of duplicating a system variable, which will have an adverse effect. It is not sufficient to check that such a variable does not already exist in the trigger, as it may be added in future software revisions.
- Any form message that you add must be assigned a number greater than 500.

Chapter 4: Reports

Introduction

Reports are constructed and modified in the *Report Generator* form and its sub-levels (*System Management → Generators → Reports*).

An easy way to create a custom report is to copy an existing one (using the *Copy Report* program in the same menu) and then make revisions to it. In fact, this method is mandatory if you want to change the sorting or grouping of columns.

Reports selectively display data stored in the database, as well as simple calculations of that data (e.g., sum totals). They can also display complex calculations of data, defined by SQL expressions. Finally, sophisticated operations can be performed on data by including the report in a procedure — i.e., a batch of executable steps that are carried out in a predefined sequence. In fact, most ***Priority*** reports are embedded in procedures. To simplify things, the examples referred to in this chapter (e.g., **ORDERSBYCUST**) will be treated as simple reports, even though some of them are actually processed reports activated by a procedure.

You can create standard and tabular reports (tables). Reports and tables can display the same data, but in different formats. Reports tend to be more detailed, displaying a relatively large amount of information. Tabular reports are used to show summarized information.

A report is characterized by:

- a unique name and title
- a set of report columns derived from the columns of one or more tables in the database
- calculated columns (optional), whose values are determined by other columns.

Note: If your system uses an SQL or Oracle database, HTML documents and reports are saved in Unicode format, using UTF-8 character encoding. If your system uses the **Tabula** database, documents and reports are saved in ASCII format.

Copying Reports

The *Copy Report* program copies:

- all report columns and their attributes
- all expressions for calculated columns
- any designated target forms
- the output title, if there is one.

It ***does not*** copy report tables, or links to procedures, menus or forms.

When assigning a name to the new report, be sure to follow the rules designated below. After the program is completed, make any needed revisions to the copy.

Report Attributes

►To revise a report's attributes (or to open a new report manually), use the appropriate columns in the *Report Generator* form.

Report Name

As with forms, the report name is a short name by which the report is identified by the system. There are certain restrictions (which also apply to report column names):

- Only alphanumeric values (uppercase and lowercase letters and digits) and the underline sign may be used (no spaces).
- The name must begin with a letter.
- You may not use a reserved word (a list of reserved words appears in the *Reserved Words* form — *System Management* → *Dictionaries*).
- The name assigned to any newly created report must include a common four-letter prefix (the same one you use for all entities that you add to ***Priority*** for the customer in question; e.g., **XXXX_ORDERS**).

Report Title

The report title is the means of identifying the report in the user interface. The designated title will appear in menus, at the top of the screen when the report is displayed, and at the top of each page of the printed report.

This title is restricted to 32 characters. You may, however, designate a longer title, which will appear on screen and in printouts, in the *Output Title* sub-level form.

Space considerations are not the only grounds for using an output title. You might also decide to use one simply to distinguish between the menu item and the report heading. For example, you might find it useful to include the word “Table” in the menu, but you would not wish it to appear in the actual report heading.

Note: If you change the report title after you have designated an output title, you will receive a warning message. This is to ensure that the output title is revised as well.

Application

Each report is assigned an application, which is used to classify reports by the type of data they access (e.g., FNC for the Financials module). If the report is copied, the application is taken from the original report. When opening a new report manually, specify a code word that aids in retrieval.

Module

Each report belongs to a given ***Priority*** module. As different modules are included in each type of ***Priority*** package, users are restricted to those reports whose modules that have purchased. If the report is copied, the module is taken from the original report. When opening a new report manually, specify "Internal Development"; this way you (and your customers) will be able to use the report no matter which modules of ***Priority*** have been purchased.

Report Column Attributes

Report columns inherit the name, title, type, width and decimal precision (in the case of **REAL** or **INT** columns) of the table columns whose data they display. With the exception of column name, all these report attributes may be modified, where desired.

►To record attributes for report columns, use the appropriate columns in the sub-level *Report Columns* form, unless otherwise designated.

When Creating a New Report

When you are creating your own report manually (and not copying an existing report), you need to decide upon the table columns that will make up the report. These columns may be derived from several different tables. The columns that make up a report can be assigned automatically or manually.

To add report columns automatically, enter the *Report Tables* sub-level of the *Report Generator* form, and specify all tables from which data are derived.

Tip: Move to the *Table Name* column and press **F6**. You will access the *Table Dictionary*, in which you can retrieve table names.

The order in which report tables are specified will affect the positions of the derived report columns. The columns of the first table to be designated will receive the first positions, those assigned to the second table will receive the next ones, and so on. Within each table, columns will be positioned in order of their insertion into the table. These column positions, which determine the order in which report columns are displayed, may be revised, where desired. You can also delete the records (from the *Report Columns* form) for those table columns which you do not need in the report, or you can hide unneeded columns.

Example: There is no need for the following columns in the **CUSTOMERS** table to appear in a report for sales orders: address, city/state, zip code, phone number, price list, internal customer number.

Note: See the rules for customizing reports, at the end of this chapter.

Adding Report Columns

To assign report columns yourself (whether to a new report or to an existing one), enter the *Report Columns* form and specify column position, column

name and table name. As with forms, the order in which columns appear in the report is determined by their relative position (an integer). Integers determining column position need not be consecutive. The column assigned the lowest integer will appear first, that with the next highest integer will appear second, and so on.

Tip: Press **F6** from the *Column Name* column to access the *Column Dictionary* or press **F6** from the *Table Name* column to access the *Table Dictionary*. The *Columns* sub-level form of the *Table Dictionary* displays all columns that belong to a given table.

Column Numbers

Each report column is identified by its unique column number, which is assigned automatically by the system. This number is used to identify the column in expressions. When report columns are added automatically (in a new report), the column with the lowest position receives column number 1, the next column is numbered 2, and so on. When columns are added to the report manually, the column number is copied from the column position. If another column already has that ID number, the new column will be assigned the next available number. Changes in column position do not affect column numbers.

Note: In order to prevent future ***Priority*** releases from overwriting any newly added columns, ***manually*** assign them a column number of at least 500 (and then change the position). For more details, see the rules for customizing reports at the end of the chapter.

Join Columns

Join columns limit the records displayed in the report so that the appropriate data is displayed. When a new report is created, if no join columns are added, the user will receive all possible combinations of data from the included table columns.

Example: If no join columns are displayed in a report based on the **ORDERS**, **ORDERITEMS** and **CUSTOMERS** tables, then the report will display Order 1000 for all customers and all ordered parts, regardless of whether there is any connection between these pieces of data. To obtain the desired combination of data – i.e., Order 1000 for Customer P600 (North Island Stars) – you have to link the order's internal customer number to the internal customer number in the **CUSTOMERS** table, by means of join columns and their tables.

Both the *Join Column* and its *Join Table* must be specified.

Note: If you have added a column to a standard report and the join is to a new table, assign a *Join ID* greater than 5. For details, see the rules for customizing reports at the end of this chapter.

Special Joins

There are two special types of joins:

- multiple joins — when two or more report columns are joined through the same table column
- outer joins — that allow for unmatched rows between the base and join tables.

*Column ID*s and *Join ID*s are used to distinguish between two joins made through the same table column. A good example of a ***multiple join*** is found in the **TOTWARHSBAL** (*Total Part Inventory*) report, which displays inventory in both standard part units and in factory units. The former unit is stored in **PUNIT** in the **PART** table and the latter is stored in **UNIT** in the same table. Both are joined to the **UNIT** table. To distinguish between the two, **PUNIT** is assigned a *Join ID* of 1 and **UNIT** is assigned a *Join ID* of 0.

Just as a distinction must be made between the two joins, so, too, must a distinction be made between the report columns that are imported through each join. For instance, the two types of units are imported from the same table column: **UNITNAME** from the **UNIT** table. The factory unit must be imported through join 0, whereas the standard unit must be imported through join 1. Thus, the former is assigned a *Column ID* of 0, whereas the latter is assigned a *Column ID* of 1.

Important note: When creating your own multiple joins, use a join ID and column ID greater than 5.

As opposed to regular joins, an ***outer join*** allows for unmatched rows between joined tables. To designate the outer join, add a question mark (?) in the relevant *Column ID* or *Join ID* column, next to the number of the ID. The decision as to where to put the question mark (column ID? join ID?) depends on where the null record is expected to be encountered. If it is in the table from which the report column is derived (i.e., the one appearing in the *Table Name* column of the *Report Columns* form), then add the question mark to the column ID. If, on the other hand, the null record is expected to appear in the join table, attach the question mark to the join ID. In the case of an additional join between the outer join table and another table, the question mark should appear in *each* of these join IDs.

Example: In the **WWWIV_1** report, which creates a header for printouts of various invoices, there is a join to the **NSCUST** table, which stores revised customer names (used mainly for walk-in customers) for all types of documents. Since not all invoices include revised customer names (but rather use the name stored in the **CUSTOMERS** table), there is an outer join between the **INVOICES** table and the **NSCUST** table via the **IV** column in both tables.
Note: As the **NSCUST** table is also used for other kinds of documents, the key for that table consists of **IV** and **TYPE**, and there is a condition on the **TYPE** column that distinguishes between invoices, orders, documents, etc. Nonetheless, it is enough to define only the **IV** column as the outer join.

Note: Outer-joined tables are accessed after regular join tables.

Report Output

You will not always wish all columns assigned to the report to be displayed during report output. For instance, there is generally no reason to display internal numbers. Hence, the **CUST** and **ORD** columns from the **ORDERS** table are not displayed in the **ORDERSBYCUST** report.

►To prevent output for a given report column, flag the *Hide* column.

User Input

►To create a parameter input screen, flag the *Input* column for each column to appear in the screen.

►If you want the input to be Boolean (Y/N) and appear as a check box, also specify *B* in the *Don't Display 0 Val* column. This only applies to a **CHAR** column with a width of one character.

To allow the user the option of defining query conditions, a parameter input screen, comprised of certain report columns, is created. While this screen can theoretically include any report columns, it is advisable only to include those columns which the user will find helpful.

Example: The input screen for the **ORDERSBYCUST** report is made up of: customer number, customer name, order number and part number. Thus, by specifying “CA001” in the *Customer* input column, the user will obtain only those orders placed by that customer.

Predefined Query Conditions

Besides creating a parameter input screen, in which the user has the option of stipulating query conditions, you can also define query conditions yourself. These conditions will hold whenever the report is run. Of course, the user can stipulate query conditions in addition to your predefined ones.

Use the *Expression/Condition* column of the sub-level *Report Column Extension* form to set query conditions. If the condition is too long to fit in that column, continue in the sub-level form, *Expression/Condition (cont.)*. Once the *Report Column Extension* form is exited, a check mark appears in the *Expression/Condition* column of the *Report Columns* form. This flag makes it easy to spot any report column with a condition.

Conditions are written in SQL and must begin with a comparative operator (<, >, <=, >=, <>, =). Only records that comply with the prescribed condition will appear in the report.

Example: The **TRANSPARTCUST** (*Customer Shipments*) report includes a condition for the **OTYPE** column from the **DOCTYPES** table: = 'C' (so as to limit the report to sales transactions, excluding purchase transactions).

Accessing a Related Form

One of the ways to input data is to access a target form (by pressing **F6** twice). Thus the target form for the *Customer Number* column would be the *Customers* form, whereas the target from the *Order Number* column would be the *Sales Orders* form. Such target forms are also accessed when the user clicks on a link within a displayed report.

As with target forms reached from forms, the target form in question must always meet the following conditions:

- it must be a root form (have no upper-level form of its own).
- its base table must include the column from which the user originated.

Generally speaking, the user accesses the default target form — the single form in the application which meets the above conditions and which shares the same name as the column's base table. However, there are several ways to override this default, so that the move will be to ***another root form based on the same table***:

- by designating a main target form (type *M* in the *Zoom/International* column of the *Form Generator* form for the form in question);
- by designating an application target form (type *Z* in the same column);
- by designating a form as a target for this particular report column (specify the name of the relevant form in the *Target Form Name* column of the *Report Column Extension* sub-level of the *Report Columns* form).

The last option overrides all other target forms, and the application target overrides the main target form.

Note: To disable automatic access from a given column, specify the **NULL** form as the target form in the *Report Column Extension* form.

Dynamic Access

Sometimes you want the target form to vary, based on the data displayed in a given record. For example, in the **AGEDEBTCUST** report, the target form of the *Invoice* column is the relevant type of invoice (e.g., **AINVOICES**, **CINVOICES**).

In order to achieve this, for the report column in question, record the following settings in the *Link/Input* tab of the *Report Columns-HTML Design* sub-level of the *Report Columns* form:

- *Link/Input Type* = P
- *Return Value Name (:HTMLACTION)* = _winform
- *Return Value Column# (:HTMLVALUE)* = the number of the column containing the **ENAME** of the target form.
  Note: The column with the **ENAME** of the target form must have a *Sort* value.
- *Internal Link Column#* = same as :HTMLVALUE above.

Accessing from a Column That is Not a Unique Key

Sometimes you want to link to a form from a report column which is not part of the unique key. For example, you may want to link from a *Part Description*

column to the *Part Catalogue* form, or from a *Details* column to the *Sales Orders* form.

In order to achieve this, for the report column in question, record the following settings in the *Link/Input* tab of the *Report Columns-HTML Design* sub-level of the *Report Columns* form:

- *Link/Input Type* = P
- *Return Value Name (:HTMLACTION)* = _winform
- *Return Value Column# (:HTMLVALUE)* = the number of the column containing the key of the target form.
  Note: The column with the key of the target form must have a *Sort* value.
- *Internal Link Column#* = leave empty.
- *Target Form (Choose)* = the name of the target form.

Writing a New CHOOSE-FIELD or SEARCH-FIELD Trigger for a Report Column

When a column is defined is an input column, if the column has a target form and that form has CHOOSE-FIELD or SEARCH-FIELD triggers, those triggers will be imported to the report input screen. You may want to write a specific CHOOSE-FIELD or SEARCH-FIELD for the report. The same restrictions that apply to form trigger names apply here as well (see Chapter 3).

►To design a new trigger, use the *Field Triggers* form (a sub-level of *Report Columns*).

Special Report Columns

You can use report columns to display special values by using the *Report Columns-HTML Design* sub-level of the *Report Columns* form. For example, you can display addresses in Google Maps, pictures or QR codes.

Displaying an Address in Google Maps

You can define a column that will appear in the report as a link to Google Maps, which will bring up the relevant address.

In order to achieve this, for the report column in question, record the following settings in the *Link/Input* tab of the *Report Columns-HTML Design* sub-level of the *Report Columns* form:

- *Link/Input Type* = Q
- *Return Value Name (:HTMLACTION)* = any value (do not leave empty)
- *Return Value Column# (:HTMLVALUE)* = the number of the column containing the address to be retrieved.
  Note: This can be a hidden column provided it has a *Sort* value.

Example: See column #60 in the **WWWORDFORM2** report.

Displaying QR Codes

You can define a column that will appear in the report as a QR code (a 2D bar code).

In order to achieve this, for the report column in question, record the following settings in the *Picture* tab of the *Report Columns-HTML Design* sub-level of the *Report Columns* form:

- *Picture* = Q
- *Width [pixels]* = Determined by the amount of data encoded
- *Height [pixels]* = Determined by the amount of data encoded
  Note: The width and height should be equal, as QR codes are square shaped and setting different values will cause image distortion.

Example: See column #190 in the **WWWIV_5** report.

Organizing Report Data

Priority enables you to organize the data displayed in reports. You can ensure that records are distinct (i.e., prevent the multiple appearance of identical records); you can sort data according to one or more columns; you can group data; you can place certain columns in a header; and you can perform certain functions upon the members of a group.

Distinct Records

Sometimes, a report will generate the same records more than once. Take the case of the **ORDDELAY** report, which displays all sales orders that meet two conditions:

- they have not been fully supplied to the customer (at least one order item is still due); and
- the due dates of those items have already passed.

Hence, an order will be displayed if the balance of at least one of its order items is greater than 0 and its due date is prior to today's date. As several items in a given order may have a balance greater than 0, the same order can appear more than once in the report. To prevent the repeated appearance, you must indicate that records should be distinct.

►Flag the *Distinct* column of the *Report Generator* form.

Sorting

Sorting data in reports is similar to sorting records in forms. Here, too, you can assign sort priorities to one or more columns, and you may designate the type of sort (in the *Report Columns* form). The records in a given report will be sorted first according to the data in the column with the highest sort priority; then according to the data for the column with the next highest sort priority; and so on. Sorts will be performed in ascending order, unless a different sort type is specified (the other options are descending sort, alphanumeric ascending and alphanumeric descending; see Chapter 3 for details).

Notes:

It is possible to sort data according to a column which does not appear during report output.

Do not change the sorting of a standard report. Instead, copy the existing report and revise the copy to suit your needs.

Grouping

Displayed columns are often organized into groups. When you group records together, you can perform certain operations on the group as a whole.

Example: A report displaying sales orders groups data by customer and order number. Thus, all orders from the same customer are displayed together, and details are presented for each order in turn. Sales totals can then be calculated for each subgroup (i.e., order) and group (i.e., customer).

The columns that define the group (order, customer) are called “Group by” columns. From one group to another, identical values are not repeated. It is therefore advisable to group by any columns which will otherwise repeat the same values. For instance, the customer name will not change until there is a new customer number. Thus, you should group by these columns. If you do not, the same customer name will be repeated for ***each*** order of this customer.

►Assign any “Group by” column an integer in the *Group by* column of the *Report Columns* form. Integers need not be consecutive, but the first “Group by” column should be assigned a value of 1. All “Group by” columns from the same table should share the same integer; they constitute a single “Group by” set. Records will first be grouped by the set with the lowest integer, then by the set with the next lowest integer, and so on.

Notes:

Grouping also affects output in tabular reports.

Do not change the grouping of a standard report. Instead, copy the existing report and revise the copy to suit your needs.

Headers

“Group by” columns can either be positioned at the leftmost side of the report, or they can be placed in report headers, at the top of each group. The latter option saves on horizontal space, enabling you to display more report columns.

If you decide to use headers, ***always*** place the first “Group by” set in the header. You can then decide whether to also include the second “Group by” set, the third set, and so on.

►To place a “Group by” column in a header, specify either *H* or *h* in the *Header* column of the *Report Generator* form. Specify a capital *H* whenever you wish to begin a new line in the header. Specify the lower case *h* when you wish to continue on the same line. Both the title of the “Group by” column (e.g., *Customer Name*) and the value of the column (e.g., CRR Holding Company) will appear in the header. If you want to display the value only, add a semicolon to the left of the revised column title (e.g., ;*Customer Name*).

Display of Grouped Records

As mentioned above, identical values are not repeated from one group to another. However, you can force repetition of identical values, if you so desire. You can also add blank lines between groups or even start each group on a new page.

►To repeat values, specify 1 in the *Repeat Group (1)* column of the *Report Columns* form for the first “Group by” set (*Group by* = 1).

►To add blank lines after the group, specify an integer (up to 10) in the *Skip Lines* column for the first “Group by” set.

►To start a new page for each group, specify “–1” in the *Skip Lines* column for the first “Group by” set.
Note: When multiple columns are included in the same "Group by" set, a new page can only be started for the first column in the set.

►To let the user decide whether to add a page break before each group, use the system variable :GROUPPAGEBREAK in the procedure that runs the report. For example, see the **PBR** input parameter in the **ACCOUNTS** procedure.

Financial Reports: Distinguishing Between Credit and Debit Balances

When you create a report that displays financial balances (such as a General Ledger report), there is a need to distinguish between credit and debit balances, by placing one within parentheses. Some users prefer to view debit balances in parentheses; others prefer credit balances. In ***Priority***, it is the value of the CREDITBAL system constant which determines whether debit or credit balances will be enclosed in parentheses. However, these parentheses will only appear in a column that is flagged as a financial balance column.

►Flag the *Balance* column of the *Report Columns* form.

Group Functions

If records have been grouped, then the data in any column which does not define the group may undergo one of several group functions (determined by the value appearing in the *Group Func.* column of the *Report Columns* form):

- Totals can be calculated for each group (*Group Func. = S* for the column to be totaled).

- Sub-totals can be calculated for any portion of the group (*Group Func. = R* for the “Group by” column in question).
- Totals can be calculated for the entire report by designating *T* in the *Group Func.* column.
- Both group totals ***and*** the entire report total can be displayed. Specify *B* instead of *S* or *T*.
- You can repeat the previous value of a string (*Group Func. = R* for the column whose data is to be repeated).
- Cumulative balances can be calculated from one line to the next, within each group (*Group Func. = A*). This is useful, for example, in a General Ledger report, which displays credit and debit balances that are updated for each displayed financial transaction. In this case, you would calculate cumulative balances for the *Balance* column.
- Complex total functions can be created for the group (*Group Func. = s, t, b* for the column to be totaled) by designating *F* in the *Col. Func* column of the *Report Columns* form. For example, see column 180 in the **INCOME_BUDGETS** report (*Budget P&L - Summary*).
  Note: The expression must reference a calculated report column that has been defined in the *Report Column Extension* sub-level form (expressions in the *Expression/Condition (cont.)* sub level form will not be taken into account). The *Column Type* of the calculated column must be **REAL**.
- A constant value (*Group Func. = C*) can be added to the cumulative balance. For instance, the value appearing in the *Opening Balance* column can be added to each figure appearing in the *Balance* column.

Note: There is one other group function (*H*) that applies only to tabular reports (see below).

Operations on Report Columns

Several functions can be performed on an individual report column. Depending on the value specified in the *Col. Func.* column of the *Report Columns* form, you can obtain:

- a sum total (*Col. Func. = S*)
- an average (*A*)
- a minimum value (*I*)
- a maximum value (*M*)
- a complex function (*F*) (defined in the *Report Column Extension* sub-level).

When you specify a column function, only the *result* of the operation is displayed. That is, values for a number of lines are compressed into values appearing in a single line. For instance, with respect to a sum total, the same information that was detailed for a given group is summarized into a single value.

Example: The **AGEDEBTCUST2** report (*Daily Aged Receivables*) makes use of the *S* column function to summarize the amounts owed (in the *Cum. Sum Outstanding* and *Sum* columns).

Note: You can combine column and group functions.

Additional Sub-totals in Reports

To include sub-totals in a report, add a hidden column with the title *#ACCTOTAL*. This will add a sub-total up to a specific point in the report.

If you want to display positive values in the report, but to treat them as negative values in the calculation of totals, add a hidden column entitled *#TOTALSIGN*. When calculating the total, all lines in the report will be multiplied by this value.

Example: See the **INCOME_STATEMENT** report.

Refining Data Display

Spacing Between Report Rows

In a report with fixed positions, you can use a hidden column with the title *#LINEHEIGHT* (**INT** type) to add a fixed amount of space between all rows of the report.

Example: See the **BTLFORM100_DET** report.

Width, Decimal Precision and Column Title

As mentioned above, report columns inherit the widths, decimal precisions and titles of the table columns whose data they display. These values may be revised, where desired.

Note: Because of this option of modifying report attributes, any changes in the width or decimal precision of a given table column ***will not affect*** existing report columns. In contrast, changes in the title of the table column ***will*** be reflected in existing reports, provided that no *Revised Title* has been assigned.

Widths are adjusted by deleting the inherited column width from the *Width* column of the *Report Columns* form and specifying the desired width. Decimal precision is adjusted by deleting the inherited decimal precision from the *Display Mode* column of the same form and specifying the revised one. However, you should be very careful when changing the display mode for an **INT** column. Finally, inherited column titles are overridden by specifying a new title in the *Revised Title* column of the same form.

Note: Whereas a decimal precision of 0 (for a **REAL** column) creates varied precision in a form, it rounds off the real number to an integer in a report. Thus, you may wish to change decimal precision for any **REAL** column with a precision of 0.

Date Displays

Dates may be displayed in a variety of formats. First of all, the user will see dates according to either the American (MM/DD/YY) or European

(DD/MM/YY) convention, depending on the language which is being used. Most examples in this manual are in American date format.

In addition, date formats are determined by the display mode and width assigned to the report column. You must ensure that there is sufficient column width for the display mode you have chosen. For instance, to display the day of the week alongside the date, specify a *Display Mode* of 1 and a *Width* of at least 12. To display time as well as date, specify a column width of between 14 and 19, depending on the display mode. Finally, you can designate a date format of your own via a calculated column (see below).

Non-display of Zero Values

You have the option of displaying or withholding the display of zero values with respect to **TIME** (00:00), **INT** (0) and **REAL** (0.0) columns. Depending on the value specified in the *Don't Display 0 Val* column of the *Report Columns* form, you can choose to:

- display zero values in all columns (default);
- leave the report column blank in case of zero values (*Y*);
- when using a group function to calculate group and/or report totals, leave both report columns and totals blank when value is zero (*A*).

Displaying HTML Text in Reports

There are several types of reports that can display HTML text:

- A fixed component report that displays only text (e.g., **WWWORD_4**).
- A fixed component report in which all fields but the text appear in the first line as “Group By” fields, and the text appears in the second line (e.g., **CUSTNOTESSUM**).
- A tabular component report in which all fields but the text appear as "Group By" fields, and the text appears in the second line under the column title (e.g., **WWWORD_2X**). In such a case, the following conditions must be met:
  - A join of the **DAYS** table with the expression DAYS.DAYNUM BETWEEN 0 AND 1.
  - A real join of the text table when DAYS.DAYNUM = 1; a join of the zero record in the text table when DAYS.DAYNUM = 0.
  - Inclusion of an expression field (width = 68) that displays the title when DAYS.DAYNUM = 0, and displays the text when DAYS.DAYNUM = 1.

HTML Design

Various options are offered for revising the default HTML design of reports. Some affect specific report columns, while others affect the entire report. This facility is particularly useful when you want to enhance the headers of printed documents (e.g., the way the company name, address and date appear in the **WWWLOGO** report) or any other one-record report. The following describes a number of the simpler, more useful options.

►To design individual report columns, use the *Report Columns-HTML Design* sub-level of the *Report Columns* form.

- *Design* tab: You can change the font and/or font color of a specific report column, as well as determine its background color. This can be defined directly in the report, or indirectly, by designating another report column whose value determines the font or color. For example, use the *Font Color Def. Column* to specify the *Col. Number* of the report column whose value sets the font color. If the designated report column is hidden, that column must have a value in the *Sort Priority* column.
- *Location* tab: The default HTML design creates a regular report, with a title at the head of each vertical column. You can use the fields in this tab to divide the page into cells and to determine how each column is positioned within a cell (for more details, see Chapter 6).
  - Indicate the number of the row(s) and column(s) in which the field will be located.
  - Designate the percentage of the *Column Width* when the field should not take up the entire cell.
  - Choose a type of *Title Design*, if desired. For instance, if you want a different font for the column title (bold type) than for the displayed value (regular type), specify *D* and define the title's font in the sub-level form.
  - Select one of the available *Data Display* options to determine how the field will be separated from the previous one. Or choose *W* to prevent line breaking of the displayed field value when there is insufficient room for all columns on the page.
  - Use the *Horizontal Align* and *Vertical Align* columns in the *Display* tab to determine how the field is positioned within its cell.
- *Picture* tab: You can use a specific report column to display a picture. Select the appropriate value in the *Picture* column and define the picture's width and height in pixels. Alternatively, you can determine the picture to be displayed based on other data appearing in the report (specify *D* in the *Picture* column and use the *Dynamic Picture Definition* column).

►To affect the design of the entire report, use the *HTML Definitions* sub-level of the *Report Generator* form.

- *Outside Border/Inside Border* tabs: Use the columns in these tabs to change the definitions of report borders, including space between boxes (columns or cells). Outside borders usually separate data between groups (e.g., in a report of orders per customer, they separate data for different customers). Inside borders usually separate data within a group (e.g., customer number and customer name).
- *More Defs* tab: You can define the number of columns to appear on each page, as well as indicate whether the report title will be displayed. The former feature allows you to display several reports next to each other (each appearing as a separate column on the page).

Example: For border definitions, see the **WWWORD_2** report. For use of the *Design* and *Location* tabs, see column #50 in the same report. For inclusion of a picture, see column #170 in that report.

Designing Reports Using CSS Classes

Priority reports are designed using predefined CSS classes and IDs, which are maintained in a system file named *style.htm* (located in the *system\html* directory). This means that it is also possible to revise the default HTML design of reports by defining additional classes and applying them to the desired HTML object.

To do so, create a copy of the existing *style.htm* file (in the same directory) with the filename *style2.htm* and use this file to define your custom CSS classes. The system will automatically add the content of this file (together with the content of the *style.htm* file) to the header of every HTML report generated by the system (i.e., inside the <head> </head> tags).

Important! Do not modify the standard *style.htm* file, as changes to this file may be overwritten by future ***Priority*** releases. Instead, make any desired changes in the *style2.htm* file only.

CSS classes can be applied to any of the following objects in ***Priority***:

- An entire report: Use the *Class* column (in the *HTML Definitions* sub-level of the *Report Generator* form) to apply a class to the entire report.
- A specific report column: Use the *Class Definition Column* column (in the *Report Columns-HTML Design* sub-level of the *Report Columns* form) to apply a class to the current report column.
- A specific font: Use the *Class* column in the *Font Definitions* form to apply a class to a specific font.

Example: For report definitions, see the **WWWTABS2** report. For report column definitions, see columns #5 and #108 in the same report.

Tips for Advanced Users

Setting the column width is particularly useful when you display two reports on the same HTML page, as it helps you line up the fields in the second report directly below the fields in the first report. To use this feature, assign corresponding columns in the two reports the same width.

To create a larger cell, place the field in question within a number of cells, indicating the range of rows and/or columns (e.g., line 2 to 4). These will then be combined into a larger cell.

To include more than one field in the same cell:

1. Assign the same row(s) and column(s) to these fields.
2. For each field involved, in the *Data Display* column, indicate how these fields should be situated (fields are ordered according to their position):
   - one underneath the other (*Y*)
   - next to each other, separated by commas (leave blank)
   - next to each other, run on (*N*).
3. For each field, in the *Title Design* column, indicate how to display its title:
   - next to the field (*Y*)
   - do not display (leave blank)

- in a separate table cell (*D*). Define the location of that cell in the next sub-level, *Column Title–HTML Design*.

Calculated Columns

In addition to report columns derived from tables, you can also create columns which display data derived from other report columns. These data are not stored in or retrieved from any database table. The value of a calculated column is determined on the basis of other columns in the report, including other calculated columns. To refer to other calculated columns in an expression, use their ***column numbers***.

Example: The *Days Late* column (#26) in the **AGEDEBTCUST2** report indicates the number of days that have passed since payment was due by comparing today's date (**SQL.DATE8** — an SQL variable) to the payment date (column #5):

```
0+ (SQL.DATE8 - (#5) > 0 ? (SQL.DATE8 - (#5))/24:00 : 0)
```

Note the use of a question mark and colon to form an if-then-else expression. Roughly, this means: if the difference between today's date and the payment date is greater than 0, then divide that difference by 24 hours (to convert it into days); otherwise, display 0.

Steps for Creating a Calculated Column

To add a calculated column to a given report, take the following steps:

1. In the *Report Columns* form, specify the position of the calculated column in the *Pos* column.
2. Designate the column's width in the *Width* column. In the case of a real number or a shifted integer, designate decimal precision as well.
3. Specify the column title in the *Revised Title* column. Note that, unlike regular report columns, which inherit titles from their respective table columns, calculated columns have to be assigned titles. If you forget to do so, the column will remain untitled when the report is run.
4. Enter the sub-level form, *Report Column Extension*.
5. Write the expression that determines the value of the column in the *Expression/Condition* column, using SQL syntax (see Chapter 2). If there is not enough room for the entire expression, continue it in the sub-level form, *Expression/Condition (cont.)*.
6. Designate the column type (e.g., **CHAR**, **INT**, **REAL**) in the **Column Type** column of the *Report Column Extension* form.

Notes:

You can quickly view all calculated columns in a given report (identified by column number and title). To do so, access the *Calculated Columns* form, a sub-level of both the *Report Column Extension* form and the *Expression/Condition (cont.)* form.

Once you exit the *Report Column Extension* form, a check mark appears in the *Expression/Condition* column of the *Report Columns* form. This flag helps you to spot any calculated columns in the report at a glance.

Displaying Alternative Date Formats

You can also use a calculated column to display dates in various formats. The default is MM/DD/YY (e.g., 01/22/92) in an American date format, or DD/MM/YY (e.g., 22/01/92) if you are using a European date format. The DTOA (**D**ate **to A**SCII) expression is used to convert dates into another pattern (e.g., *Fri, May-12-06*; *12 May 2006*). It is used as follows:

DTOA(table.column, 'pattern')

Example: DTOA(ORDERS.CURDATE, 'MMM DD, YYYY') converts an order date of 07/12/06 to Jul 12, 2006 (in American format). The type of such a calculated column would be **CHAR**, and its width would be 12.

Note: For a list of available DTOA patterns, see Chapter 2.

Condition for a Calculated Column

Sometimes you may wish to include a calculated column that has a condition. In that case, you do not create a calculated column, as described above. Instead, add a dummy column to the form (column name = **DUMMY**; table name = **DUMMY**) and assign ***it*** the desired condition. The condition itself must be preceded by "**=**1 AND" or "= DUMMY.DUMMY AND".

Example: Column #131 in the **ACCBYFNCPAT** (*Account Transactions by Type*) report includes the following expression for **DUMMY.DUMMY**:
= 1 AND (FNCITEMS.DEBIT1 <> 0.0 OR FNCITEMS.CREDIT1 <> 0.0)
That is, the report only displays a sum when either **DEBIT1** or **CREDIT1** is greater or less than zero.

Conditions in a *Group by* Column

Suppose you want to create a report that displays the number of sales orders in a designated time period for each customer, but only for customers with more than a designated number of sales orders. In order to achieve this, you can specify a search condition (e.g., HAVING COUNT(*) > :MIN) for the group:

1. Add a dummy column to the report (column name = **DUMMY**; table name = **DUMMY**).
2. Hide the new report column.
3. Indicate the function to be performed on the column (e.g., *Col. Func. = S*).
4. In the *Report Column Extension* sub-level form, write an expression that represents the desired condition, using SQL syntax.

In the above example, you would record "= 0 AND COUNT(*) > :MIN", where :MIN is an input variable received by the procedure that executes the report.

If you then dump the report's query using the *SQL Development* (**WINDBI**) program, you will see that the SQL query now includes the following conditions in the GROUP BY clause:

```
HAVING SUM(DUMMY.DUMMY) = 0
AND COUNT(*) > :MIN
AND (1 = 1)
```

Of course, the first and last conditions are always true.

Using a Complex Function

Sometimes you will want to perform a complex operation on a calculated column (beyond a simple sum, average, minimum or maximum).

Example: The **TOTALTRANSBAL** (*Inventory Movement in Period*) report includes a complex function in column #81 (*Avg Monthly Consumption*):
(SUM(#56) / (#80))
Column #56 displays *Outgoing Transact'ns*, while column #80 calculates the number of months in inventory in the period.

Complex functions are defined (like all calculated columns) in the *Report Column Extension* form. In addition, a *Col. Func.* of type *F* must be specified.

Tables (Tabular Reports)

Creating a Tabular Report

Reports can also be displayed in tabular form, divided into columns and rows. Such tables succinctly summarize report data. To define a report as tabular, specify *T* in the *Type* column of the *Report Generator* form.

Each tabular column displays data for the report column which has been assigned a graphic display value of *X* (in the *Graphic Display* column of the *Report Columns* form).The order in which these data appear is determined by a non-displayed column (e.g., the internal part number) that has been assigned a graphic display value of *O*. This is particularly useful when the *X* data display dates. If you have not assigned *O* to any column, then the *X* data will be sorted alphanumerically. Finally, the content of the table cell is determined by the report column (or columns) which has been assigned a graphic display value of *T*.

Example: In the **AGEDEBT_T_C** report, which displays monthly customer aging data, the X column displays the month (Jan-06, Feb-06, etc.). In order to sort correctly by month number, rather than month name, a hidden sort column (type *O*) is used (see column #102).

Notes:

Each group appears in a separate row in the table.

You can save horizontal space by using vertical mode (specify *V* in the *Table Display Mode* column of the *Report Generator* form).

If only one row is displayed for the group in question, you can to choose to hide its title (specify *h* in the *Table Display Mode* column).

Totals

The table can display row totals, sum totals (per row and group) and/or grand totals (for all groups and all rows). This is achieved by assigning a group function to the column in question, as follows:

- If a row total (group function = *H*) has been specified, each row will be totaled.
- If a sum total (group function = *S)* has been specified, each group and row will be totaled.
- If a total (group function = *T*) has been specified, column totals will appear.
- If both a sum and a total (group function = *B*) have been specified, the table will display row totals, group totals, column totals and a grand total.

Multi-Company Reports

You can define a report to display data from multiple companies (databases). In order to create such a multi-company report, add the following columns to the report:

- A displayed column, with a *Column Name* of TITLE and a *Table Name* of ENVIRONMENT.
- A hidden column, with a *Column Name* of DNAME and a *Table Name* of ENVIRONMENT. Its *Expression/Condition* should be: = SQL.ENV

Example: See the **ORDERSREP** report.

Note: The companies displayed in the report are the ones the user has selected via *Define Multiple Companies* (in the *File* menu).

Running a Report

A simple report (one which is not part of a procedure) may be run in one of three ways:

- directly by means of a program
- via the menu to which the report is attached
- via the form to which it is linked.

Using a Program to Run the Report

You can use the *Run Report* program (accessed from the *Reports* menu) or the *Run Report/Procedure* program (run by Direct Activation from the *Report Generator* form) to run a report. This method is particularly useful during the report's development stages.

Creating Menu Links

Reports, like forms, are generally accessed from menus. To position a given report within a menu, take the following steps:

1. Enter the *Menu/Form Link* form, a sub-level of the *Report Generator* form.
2. Indicate the name of the menu in question and specify *M* in the *Type* column. The title of the menu in question will appear automatically, to verify that you have specified the correct menu name.
3. Specify an integer to determine the order of the report within the menu.
4. Disregard the *Background Execution* column. This is only relevant for reports activated from within a form.

Note: The *Menu Items* form, a sub-level of the *Menu Generator*, serves a similar purpose.

Direct Activation from a Form

Like most other ***Priority*** entities, reports may be run directly from within a form. This option may be offered in addition to or in place of accessing the report through the menus. By linking a report to a form, it can be run and printed out directly from the form in question. When directly activated from the form, input will be restricted to the form record on which the cursor rests. Only columns which belong to the form's base table serve as input data.

Direct Activation of a report from within a form can take place in either the foreground or the background. If it is in the background, the user will be able to continue work in the form while the report is being run. Indeed, printouts of reports are generally run in the background.

To allow for Direct Activation of a report from within a form, take the following steps:

1. Enter the *Menu/Form Link* form, a sub-level of the *Report Generator* form.
2. Indicate the name of the form in question and specify *F* in the *Type* column. The title of the form in question will appear automatically, to verify that you have specified the correct form name.
3. Specify an integer to determine the order of the report within the form's list of Direct Activations.
4. If the report is to be run in the background, flag the *Background Execution* column.

Note: The *Direct Activations* form, which is a sub-level of the *Form Generator* form, serves a similar purpose.

Processed Reports

A procedure is a batch of executable steps that are carried out in a predefined sequence. One of the steps in a procedure may be the processing of report data. Thus, any given report may be part of a procedure. Include a report step in a procedure when the report requires manipulations other than simple averages, sum totals, minimums or maximums. The procedure allows for more complex operations to be performed on data. A detailed explanation of procedures can be found in Chapter 5. Generally, you include a report in a procedure via the *Procedure Steps* form, a sub-level of the *Procedure Generator* form. You can then view that linkage in the *Procedure Link* form, another sub-level of the *Report Generator* form.

Help Messages

You can use the *Report Generator* to create on-line help messages for an entire report and/or any report column appearing in its parameter input screen. Help for the report itself is specified in the *Help Text* form, a sub-level of the *Report Generator* form; input column help is designated in the *Help Text* form, a sub-level of the *Report Columns* form. Whenever help messages (for the entire report or for any input column) are added or modified, the date and time of this revision appear in the *Help Date* column of the *Report Generator* form.

Note: Do not write help for any report which is activated from a procedure; instead, write the help for the procedure.

Line Breaking

Line breaking for help messages is set automatically. Therefore, you can freely edit the text in the *Help Text* form without having to worry about the length of lines. You should therefore refrain from the use of hyphenation. To force the beginning of a line, use the code \n.

Example: The help for the **OPENORDIBYDOER** report says:
Run this report to view orders to carry out, sorted by due date. \n
\n
The report displays the name of the user assigned to each order.

Referring to Other Entities

Often, you will want to refer to other ***Priority*** entities in your help text. As entity titles are easily changed (even by the user), whereas their names are relatively fixed by the time you begin to create help messages, ***Priority*** offers a mechanism for referring to an entity by *name* in the help text; its current *title* will consequently be displayed when the help is called up by a user.

To refer to a specific entity in a help message, designate the entity name followed by a period and its type, where *F*=form, *R*=report and *P*=procedure, enclosing all of this in curved brackets.

When your help messages are more or less complete, you should check that all designated entity names and types are correct. To do so, run the *Check Entity Names in Msgs/Help* report (*System Management → Dictionaries → Messages and Help Text*). Any mistakes that are uncovered will have to be corrected in the appropriate *Help Text* form.

Rules for Customizing Reports

- If you revise a standard report, you must follow some rules to ensure that your customizations are not overwritten by future ***Priority*** releases:
- Any columns that you add to a standard report must have an internal number (*Col. Number*) greater than 500. After assigning the column number, you will probably need to fix the position of the column.
- If you add a join to a new table, assign a *Join ID* greater than 5.
- Do not change the sorting or grouping of standard reports. If you have to do so, copy the standard report and create one of your own.
- You cannot delete a standard column from a standard report.
- Include the appropriate four-letter prefix (e.g., **XXXX_ORDERS**) in the name of any report you create.
- When creating your own multiple joins, use a join ID and column ID greater than 5.
- Whenever you revise a standard report or write a new one, it is imperative to check the report's optimization. To do so, run the *SQL Development* (**WINDBI**) program (*System Management → Generators → Procedures*). From the **Optimization** menu, select **Report Optimization** and record the internal name of the relevant report.

Chapter 5: Procedures

Introduction

Procedures are a set of executable steps carried out in a predefined order. They are often used to create ***processed reports*** — reports that are generated following data manipulation. Similarly, they can be used to create a ***document***. This is a special procedure that collects data from more than one report and displays a final file using an Internet browser (see Chapter 6). Another type of procedure is used to create a new document (e.g., a sales order) on the basis of another document, by means of an interface to a form. Finally, procedures also serve to run the SQLI program as well as internal programs, which perform data manipulations and other tasks.

You should not create a procedure that inserts records directly into a table. Instead, use the procedure to run an interface to a form that will insert the records. For details, see Chapter 7.

A procedure is characterized by:

- a unique name
- a title
- steps of execution (the entities/commands that are run)
- parameters.

Procedures are constructed and modified in the *Procedure Generator* form and its sub-levels (*System Management → Generators → Procedures*).

You cannot customize an existing procedure. Rather, you must copy it (using the *Copy Procedure* program in the same menu) and make revisions to the copy. The two procedures (standard and customized) should be very similar in terms of their logic.

Note: There are some slight differences when procedures are written for the ***Priority*** web interface; see Chapter 15 for details.

Copying Procedures

The *Copy Procedure* program copies:

- all the procedure steps
- all their parameters
- all step queries and procedure messages
- any designated target forms.

It ***does not*** copy the output title, links to menus, forms or other procedures.

When assigning a name to the new procedure, be sure to follow the rules designated below. After the program is completed, make any needed revisions to the copy.

Notes:

If the procedure creates reports, you may need to copy one or more of those reports as well, using the *Copy Report* program. For details, see Chapter 4.

If the procedure runs a program, you must not make any changes to parameters used by the program.

Procedure Attributes

►To revise a procedure's attributes (or to open a new procedure manually), use the appropriate columns in the *Procedure Generator* form, unless otherwise designated.

Procedure Name

The procedure name is a short name by which the procedure is identified by the system. The following restrictions apply:

- Only alphanumeric values (uppercase and lowercase letters and digits) and the underline sign may be used (no spaces).
- The name must begin with a letter.
- You may not use a reserved word (a list of reserved words appears in the *Reserved Words* form, at: *System Management → Dictionaries*).
- The name assigned to any newly created procedure must include a common four-letter prefix (the same one you use for all entities that you add to ***Priority*** for the customer in question; e.g., **XXXX_WWWSHOWORDER**).

Procedure Title

The title is the means of identifying the procedure in the user interface. The procedure title will appear in menus and at the top of any report that is output by the procedure. Procedure titles are restricted to 32 characters. You may, however, designate a longer title, which will appear in printouts, in the *Output Title* sub-level form. For a procedure that runs reports, specify the output title for the report instead (using the appropriate form in the *Report Generator*). If you specify an output title for both, then the one assigned to the report will be used.

Procedure Type

Designate the type of procedure:

- If the procedure runs a report, specify *R* in the *Rep/Wizard/Dashboard* column.
- If the procedure runs a report but you want to prevent the Print/Send Options dialogue box from appearing to the user, specify N in the *Rep/Wizard/Dashboard* column.
- If the procedure creates a document, specify *R* in the *Rep/Wizard/Dashboard* column and *Y* in the *HTML Document* column.

Note: Full details on processed reports appear later in the chapter. Details on documents can be found in Chapter 6.

Application

Each procedure is assigned an application, which is used to classify procedures by the type of data they access (e.g., FNC for the Financials module). If the procedure is copied, the application is taken from the original procedure. When opening a new procedure manually, specify a code word that aids in retrieval.

Module

Each procedure belongs to a given ***Priority*** module. As different modules are included in each type of ***Priority*** package, users are restricted to those procedures whose modules they have purchased. If the procedure is copied, the module is taken from the original procedure. When opening a new procedure manually, specify “Internal Development”; this way you (and your customers) will be able to use the procedure no matter which modules of ***Priority*** have been purchased.

Procedure Steps

►To define the entities/commands in a procedure, and the order in which they are accessed, enter the *Procedure Steps* form, a sub-level of the *Procedure Generator* form.

A procedure is composed of a set of entities and/or commands that are executed in a fixed order. Each entity, identified by its name and type, constitutes a separate step in the procedure. The order of execution is determined by the *Step* column.

Step Types

There are several different types of procedure steps, each of which is a valid entity:

- a report (*R*), which generates a report after data processing
- a form (*F)*, used to input data
- a procedure (*P*), which activates a sub-procedure
- a Basic command (*B*), used for parameter input, message output and flow control
- a form load interface (I), used to load data into a ***Priority*** form (see Chapter 7)
- a table load file (*L*), used to import external data into the application (see Chapter 7)
- a compiled program (type *C*), used to manipulate data.

Specify the name of the entity or Basic command that constitutes each procedure step (in the *Entity Name* column), as well as its type.

Basic Commands

The following is a list of useful Basic commands. Examples of their usage appear throughout the chapter.

- **BACKGROUND** (*Background Execution)* — Causes the remainder of the procedure to be run in the background.
- **CHOOSE** (*Select Parameter*) — Creates a menu of exclusive options, one of which must be chosen by the user (by flagging one of the radio buttons). This Choose menu will ***not*** be displayed if the procedure is run by Direct Activation from a form.
- **CHOOSEF** (*Select Parameter*) **—** Same as CHOOSE, except that the menu ***will*** be displayed when the procedure is run by Direct Activation.
- **CONTINUE** (*Continue*) — Opens a pop-up menu of two exclusive options, one of which must be chosen by the user. The procedure will continue if the **OK** option is chosen; it will halt if the **Cancel** option is selected. This pop-up menu will ***not*** be displayed if the procedure is run by Direct Activation from a form.
- **CONTINUEF** (*Continue*) — Same as CONTINUE, except that the menu ***will*** be displayed when the procedure is run by Direct Activation.
- **END** (*End of Procedure*) — Ends execution of the procedure; generally used in conjunction with the GOTO command.
- **GOTO** (*Jump to Step*) **—** Causes a jump to a designated procedure step (e.g., to repeat the procedure, or a portion of it, following a CONTINUE command).
- **HTMLCURSOR** (*Create HTML Document*) **—** Declares the cursor for a document. This step first creates a linked file that holds the records selected in the PAR input parameter.
- **HTMLEXTFILES** (*Attach Files*) — Causes the program that prints a document to include a flag in user input which allows the user to print attachments (stored in a sub-level of the document) as well.
- **INPUT** (*Parameter Input*) — Inputs parameter values; in the case of user input, creates a parameter input screen. The input screen will ***not*** be displayed if the procedure is run by Direct Activation from a form. In document procedures, this command is also used (as a final step) to display the document (see Chapter 6).
  Note: User input can also be defined for the **SQLI** program, as well as in specific report columns in a processed report (see more below).
- **INPUTF** (*Parameter Input*) — Same as INPUT, except that the parameter input screen ***will*** be displayed when the procedure is run by Direct Activation.
- **MESSAGE** (Message) — Displays a procedure message on screen. The message number is stored in an **INT** parameter and the message content is recorded in the *Procedure Messages* form. This message will ***not*** be displayed if the procedure is run by Direct Activation from a form.
- **MESSAGEF** (Message) — Same as MESSAGE, except that the message ***will*** be displayed when the procedure is run by Direct Activation.
- **PRINT** (*Print Message*) **—** Displays on screen the contents of a file. Execution of the procedure continues after the user confirms receipt of the message. If the file is empty or does not exist, execution of the procedure

continues uninterrupted. Alternatively, this command displays a designated string of characters. This message will ***not*** be displayed if the procedure is run by Direct Activation from a form.

- **PRINTF** (*Print Message*) **—** Same as PRINT, except that the message ***will*** be displayed when the procedure is run by Direct Activation.
- **PRINTCONT** (*Print Message & Continue/Stop*) — Like the PRINT command, displays on screen the contents of a file, but also offers the user the options of continuing execution of the procedure or halting. This message will ***not*** be displayed if the procedure is run by Direct Activation from a form.
- **PRINTCONTF** (*Print Message & Continue/Stop*) **—** Same as PRINTCONT, except that the message ***will*** be displayed when the procedure is run by Direct Activation.
- **PRINTERR** (*Print Error*) — Displays on screen the contents of a file containing an error message and causes procedure failure. If the file is empty, or no file exists, the procedure continues uninterrupted.
- **SHOWCOPY** (*Create Certified Copy*) – Creates a certified copy of a document. The command takes two parameters: the document ID and the document type (for a financial document, IV and 'I'; for an inventory document, DOC and 'D'). It must be followed by an END step and a BACKGROUND step. See, for example, the **IVSHOWCOPY** procedure. For more on certified copies, see Chapter 6.
- **URL** (*Open Webpage*) — Opens a webpage according to a web address stored in an ASCII file.
- **WRNMSG** (*Warning Message*) — Like the MESSAGE command, displays a procedure message on screen. The difference is that a **Cancel** button appears as well, allowing the user to halt execution of the procedure. This message will ***not*** be displayed if the procedure is run by Direct Activation from a form.
- **WRNMSGF** (*Warning Message*) **—** Same as WRNMSG, except that the message ***will*** be displayed when the procedure is run by Direct Activation.

Procedure Parameters

►To define parameters, use the *Procedure Parameters* form, a sub-level of the *Procedure Steps* form.

Most procedure steps incorporate parameters. These are a means of transferring arguments from the procedure to programs or reports activated by the procedure. They also serve to transfer information from one procedure step to another (or from the user to the procedure step).

Example: When a program finds an error, it writes the appropriate message to a file (i.e., a parameter in the program). The parameter is then passed on to the PRINTERR command, which prints out its contents on screen.

Only certain steps require parameter specification (e.g., CONTINUE and END do not). Moreover, different types of steps require that different attributes of the parameter be defined. For instance, you need not specify type for any

parameter in a CHOOSE or CHOOSEF command, and there is no need to designate position for any parameters in a report step.

Parameter Name and Title

All procedure parameters must be given a name. This must meet the same requirements as the procedure name, with two exceptions: (1) there is no prefix); and (2) the parameter name is restricted to up to three characters (e.g., DAY, MSG, AA).

The title is a short description of the parameter. When the parameter is for input purposes or generates a Choose menu, the user will see the title. It can also be used to store a brief message that is displayed by the PRINT, PRINTCONT or PRINTERR command.

Parameter Order

The order of parameters in a given procedure step is determined by their position (an integer). Integers need not be consecutive; the parameter with the lowest integer will be first. In a program, parameter position will determine the order of parameters that are passed to the procedure.

Note: If the procedure step consists of a single parameter, it is not necessary to specify position at all. Nor is it necessary to indicate position for parameters in a report step.

Parameter Content

There are three distinct types of procedure parameters:

- Those which are constants or variables.
- Those which are text files.
- Those which are linked files.

A parameter may be assigned a constant value or a variable in the *Value* column. The value of any variable, which is identified by the prefix ":" (e.g., :date), must be specified earlier in the procedure.

The content of a text file is determined by a step in the procedure (generally by a program). This is generally a message file, which will then be printed out by a PRINT, PRINTCONT or PRINTERR command.

Note: If your system uses an SQL or Oracle database, all temporary text files created by the system (e.g., procedure message files) are saved in Unicode format. If your system uses the **Tabula** database, documents and reports are saved in ASCII format.

A linked file is a copy of a database table that contains only certain records. This file includes all the table's columns and keys. The content of a linked file is input by the user via a parameter input screen or by means of a form. Such a file is tied to a specific database table and column by means of the *Table Name* and *Column Name* columns. The user then retrieves specific records by specifying search criteria for the column in question, by moving from the column in the parameter input screen to a target form and retrieving desired

records, or by retrieving data from the form that opens instead of the input screen (as a result of a Form step in the procedure). A third possibility is to input data from a specific form record. This occurs when the procedure is directly activated from within a given form. Whatever method is used, a file of records is created. Data manipulations may then be carried out on the linked file.

Parameter Type

With the exception of parameters for a CHOOSE or CHOOSEF command, all procedure parameters must have a specified type.

- A parameter which contains a constant value must be assigned one of the valid column types (**CHAR, REAL, INT, UNSIGNED, DATE, TIME, DAY**).
- A parameter that is a text file must be assigned **ASCII** type.
- A parameter that stores text must be assigned **TEXT** type.
- If the parameter is a linked file of records, you must designate one of the following types and specify the names of the table and column to which the file is linked:
    - Select **FILE** if the linked file comprises a group of records input from the database.
    - Select **NFILE** if the linked file comprises a group of records and you want the link table to remain empty when the user enters * or leaves the field empty.
    - Select **LINE** if the file consists of a single record input from the database.

User Input

When the values of parameters are determined by the user, an *I* is specified in the *Input* column (or an *M*, if the input is mandatory). Input does not have to be defined via an INPUT command. You can also specify input in a CHOOSE command, a form step or an SQLI step. In addition, in a processed report, columns can be flagged for input in the report itself (by flagging the *Input Column* in the *Report Columns* form).

Inputting a New Value

If, during input, the user is to specify new values (not in the database), you must record a parameter (of any type except **ASCII**, **FILE** or **LINE**), together with a valid width and a title. In such a case, an equal sign (=) will appear in the first line of the column in the parameter input screen.

You can insert a pre-set, revisable value into the input screen — for instance, the current date (SQL.DATE8) for a **DATE** parameter. The user can then choose between using that value or modifying it. To do so, specify the pre-set value in the *Value* column (of the *Procedure Parameters* form) for the parameter in question. Of course, the designated value must match the parameter's data type. Alternatively, you can specify the name of a variable (e.g., :date),

provided that its value has already been defined (e.g., in a previous procedure step).

The pre-set input value appears the first time the user runs the procedure. If that value is then revised, the revised value will appear the next time the user runs it. You can, however, ensure that the pre-set value ***always*** appears. To do so, enter the *Procedure Parameter Extension* sub-level form and specify *d* in the *Type* column.

If, instead, you want the input to be Boolean (Y/N) and appear as a check box, specify *Y* in the *Type* column (of the *Procedure Parameter Extension* form).

Choosing Between Several Fixed Options

If you want the user to choose between several predefined options within a given input field, create a parameter of **INT** type, and record the various options in consecutive messages (in the *Procedure Messages* form). Then enter the *Procedure Parameter Extension* sub-level form and specify *C* in the *Type* column. Indicate the range of message numbers in the *From Message* and *To Message* columns.

Notes:

To use messages recorded for a different procedure, designate its name in the *Entity Name* column.

The value of this parameter can then be included in subsequent procedure steps (e.g., as the value assigned to a GOTO command).

Choosing Options from a List of Radio Buttons

Alternately, you can create a separate input screen of options, in which the user must flag one of the radio buttons. To do so, use the CHOOSE command. The first parameter of a CHOOSE command stores the result of the user's choice. Its title appears at the top of the pop-up menu, as its heading. It is this parameter that may be included in subsequent procedure steps.

There are two methods for creating the radio buttons: you can define a list of additional parameters or write a CHOOSE query.

Method 1: When you define a list of parameters, assign each one a unique constant value (an integer), a title and a position. Titles will appear next to the radio buttons in the order dictated by each parameter's position. The value of the option (i.e., parameter) chosen by the user will be assigned to the first parameter.

Method 2: You can write a CHOOSE query in the *Step Query* sub-level form of the CHOOSE/CHOOSEF procedure step. This is a regular SQL query with three arguments in the SELECT clause. All arguments must be of **CHAR** type (convert numbers to strings using the ITOA function; see Chapter 2).

The first two arguments in the CHOOSE query are displayed next to the radio button and the third is the value to be assigned to the parameter. If you want

to display a single value as a description next to each radio button, use the empty string (' ') as the second argument.

Example: See the **COPYPRICELIST** procedure.

Note: Rules for the CHOOSE query are similar to those in CHOOSE-FIELD triggers.

Retrieving Records Into a Linked File

Another type of user input involves retrieval of records from a given database table into a linked file. This case requires specification of *Column Name* and *Table Name*. The user can then specify a search pattern in that column, or access a form which displays the records in that table and then retrieve desired records.

Inputting Text Into an HTML Screen

When the procedure parameter type is **TEXT**, the user keys in an unlimited number of lines in the text field, and these lines are returned to the procedure via a file linked to the **PROCTABLETEXT** table.

Example:

```
LINK PROCTABLETEXT TO :$.TXT;
GOTO 99 ERE :RETVAL <= 0;
INSERT INTO GENERALLOAD (LINE,TYPE,TEXT)
SELECT 2+KLINE, '2',TEXT FROM PROCTABLETEXT WHERE KLINE > 0;
UNLINK PROCTABLETEXT;
```

Other Input Options

Additional columns in the *Procedure Parameter Extension* sub-level form also affect user input:

- To allow users to input a file attachment, specify *Y* in the *Browse Button* column. A Windows Explorer will open in which the user selects the file in question.
  Note: The parameter in question must be of **CHAR** type.
- To allow users to save a new file, specify *S* in the same column.
- To encode user input (e.g., when a password is given), flag the *Hide User Input* column. Anything recorded by the user will appear as a row of ++++++ marks.

Writing a New CHOOSE-FIELD or SEARCH-FIELD Trigger for a Procedure Parameter

When a parameter is defined as an input column, if the column has a target form and that form has CHOOSE-FIELD or SEARCH-FIELD triggers, those triggers will be imported to the input screen.

You can also write a specific CHOOSE-FIELD or SEARCH-FIELD for the procedure. Your trigger can contain references to any input value specified by the user within the same procedure step. For instance, if the procedure step

contains an input parameter called CST, its value will be stored in the :PROGPAR.CST variable. This is useful, for example, if a given procedure step contains an input column for a *Sales Rep* and another input column for a *Customer*, and you want the Choose list for the latter column to display only those customers that are associated with the specified sales rep.

The same restrictions that apply to form trigger names apply here as well (see Chapter 3).

►To design a new trigger, use the *Field Triggers* form (a sub-level of *Procedure Parameters*).

Accessing a Related Form

When the parameter is a linked file, the user can specify an exact value for that database column, stipulate a query pattern for that column or access a related target form in which to retrieve records.

In the last case, the user normally arrives at a default target form. This is the form that serves as a "window" into the database table to which the column in question belongs (i.e., the table specified in the *Procedure Parameters* form), provided that the table and form share the same name. However, there are several ways to override this default, so that the move will be to ***another root form based on the same table***:

- by designating a main target form (type *M* in the *Zoom/International* column of the *Form Generator* form for the form in question);
- by designating an application target form (type *Z* in the same column);
- by designating a form as a target for this particular parameter (specify the name of the relevant form in the *Target Form Name* column of the *Procedure Parameter Extension* form).

The last option overrides all other target forms, and the application target overrides the main target form.

Example: Specify a target form for the parameter when there is more than one form linked to the same base table (e.g., **PART** and **LOGPART**).

The specified target form must meet two conditions:

- It must be a root form (have no upper-level forms of its own)
- Its base table must include the column from which the user originates.

Note: To disable automatic access from a given column, specify the **NULL** form as the target form.

Input During Direct Activation

When a procedure is activated from within a form, input is received from the record on which the cursor rests. That is, a linked file is created, based on the form's base table and consisting of that single record. This linked file is input to the procedure by the PAR parameter. Therefore, any procedure that is activated from a form must meet the following conditions:

1. The PAR parameter must be in the first position of the procedure's first step.
2. It must be of **FILE** type.

If you want to receive additional input from the user, you must create an input screen (using the INPUTF command) or a menu of choices (using the CHOOSEF command).

Tip: To run the same procedure from a menu, make sure that *Column Name* and *Table Name* are also recorded.

Using a Form for Input

The content of a linked file may also be input by means of a form (procedure step of type *F*). This form must be the root of a form tree. It is loaded together with all the sub-level forms in its form tree. A form step is useful when you wish to allow the user to retrieve several records that will serve as input, particularly when the query is complex (entailing several conditions) or when retrieval is not through a key.

Example: See the **CLOSEAIVS** procedure.

Step Queries

►To record SQL statements, use the *Step Query* form, a sub-level of the *Procedure Steps* form.

INPUT, SQLI, HTMLCURSOR and HTMLEXTFILES commands can be accompanied by SQL statements that serve as step queries. Such queries are carried out ***after*** parameter input.

Notes:

SQLI is a procedure step that executes SQL statements.

HTMLCURSOR and HTMLEXTFILES are Basic commands used in creating documents (see Chapter 6).

Error and Warning Messages

The SQL statements in step queries resemble those in form triggers. For instance, they can generate error or warning messages (by means of ERRMSG and WRNMSG statements). The contents of these messages are generally specified in the *Procedure Messages* form, a sub-level of the *Step Query* form. Long messages (taking up more than one line) can be continued in the next sub-level, *Procedure Messages (cont.)*. Alternatively, message content can be taken from an external file (see Chapter 3).

In any given error or warning message, you can include a message parameter (<P1>, <P2>, <P3>). The values to be assigned to these parameters are defined in the query that generates the message, by means of the system variables :PAR1, :PAR2 and :PAR3.

You can also refer to a specific ***Priority*** entity in the message, using the format **{*entity_name*.**{ **F** | **R** | **P** } **}**, where F = form, R = report and P = procedure. That is, you designate the entity name and type, and the entity's title will appear in their place. This format is useful because entity names are rarely changed, whereas titles are rather likely to be modified in upgraded or customized versions. In this way, the most up-to-date title will appear in your message.

You can also send a mail message via a step query, using the MAILMSG command; for details, see Chapter 3.

When activating the MAILMSG command from an SQLI step in a procedure, messages are not actually sent until the SQLI step is completed. In the interim, the MAILMSG command stores any messages being sent in that SQLI step in a buffer. This buffer is limited to 100 messages, meaning you cannot send more than 100 messages in the same SQLI step.

If you wish to send more than 100 messages, you can bypass this limit by creating an internal loop between procedure steps, using the GOTO Basic command. Finish the SQLI step and, in the next procedure step, use the GOTO command to return to that SQLI step (or to continue to the next procedure step, once all messages have been sent). In each iteration, up to 100 messages will be sent.

Note: Use this option carefully and avoid creating an infinite loop.

Parameter Variables

Step queries can include SQL variables that refer to specific parameters. The system defines an SQL variable for each parameter, which stores its value (:*ProcedureName.ParameterName*). Similar to the case of form variables, the procedure name can be replaced by the wildcard "$" if reference is to the current procedure (i.e., :$.*ParameterName*).

Procedures With Heavy Processing

In a procedure that requires heavy processing via a cursor, you can display a progress bar that indicates to the user how far the program has progressed. To do so, use the following step query:

```
DECLARE mycursor CURSOR FOR …;
OPEN mycursor;
:N = :RETVAL; :I = 0;
GOTO 9 WHERE :N <= 0;
LABEL 1;
FETCH mycursor INTO …;
GOTO 8 WHERE :RETVAL <= 0;
:I = :I + 1;
DISPLAY :I OF :N;
{processing of current record}
LOOP 1;
LABEL 8;
CLOSE mycursor;
LABEL 9;
```

Important! You can use the same cursor more than once within a given procedure or form trigger, but the declaration can be done only once. If, for example, you write a buffer that contains a cursor, and you want to use that buffer more than once in a procedure, you must write the declaration section in a separate buffer.

Checking SQL Syntax

You can check the SQL statements in the step query for syntax errors, prior to activation of the procedure itself, by running the *Syntax Check* program by Direct Activation from within the *Procedure Generator* form.

Tracking Changes in Step Queries

You can track changes to step queries once they have been included in prepared version revisions. For details, see Chapter 9.

Flow Control

The CONTINUE, GOTO and END commands (usually together with some type of Choose option) all affect procedure flow. In brief, they serve the following purposes:

- **CONTINUE** — enables the user to continue or exit the procedure (see also the PRINTCONT command)
- **GOTO** — jumps to another procedure step
- **Choose option —** the value of the selected option is passed on to the GOTO command, determining the step to which the GOTO jumps
- **END —** ends the procedure.

Continuing/Halting the Procedure

When a procedure involves heavy data manipulation or has far-reaching effects on the database, you may wish to offer the user the option of exiting it prior to data processing. The user may have accidentally activated the

procedure, or may simply have changed his or her mind. The CONTINUE command (without parameters) is used for that precise purpose.

Using the GOTO Command

The GOTO command, which is useful in conjunction with CONTINUE, causes the procedure to jump forwards or backwards to another procedure step. The GOTO command always has a single parameter whose value is ***the procedure step at which to continue*** and whose type is **INT**.

Note: The value of the GOTO parameter may be a constant designated in the *Value* column of the *Procedure Parameters* form; it may be determined by an SQL statement; or it may be determined by the user's choice of one of the CHOOSE options.

Activating a User-Chosen Option

You can create a procedure that offers the user several options and then activates certain subsequent procedure steps based on the chosen option. To design such a procedure, use the Choose functionality (via an input step of Choose items or by means of the CHOOSE command) together with GOTO and END commands. That is, the value of the chosen option can be used to determine the value of the GOTO command. Simply use the same parameter name for the GOTO command and leave the *Value* column blank (remember to specify a type of **INT**). Thus, if the user chooses the option whose value is 60, the procedure will proceed at step 60. It will continue until an END command is encountered (or until there are no more procedure steps).

You can also include a Choose option that simply ends the procedure, in case the user wants to change his or her mind. This is achieved by jumping to an END command (e.g., step 50). Note that the END command has no parameters.

Message Display

PRINT, PRINTCONT and PRINTERR

The PRINT, PRINTCONT, and PRINTERR commands are very similar. However, whereas the PRINT command only displays a message, the PRINTCONT command also allows the user the options of continuing execution of the procedure or stopping. In this sense, it is similar to the CONTINUE command. PRINTERR command causes procedure failure; thus, it should be used to display error messages which explain that the procedure cannot be successfully completed.

Printing a Fixed Message

In addition to their usage for printing messages from a message file, the PRINT and PRINTCONT commands can be used to print out a fixed message on screen. To use the PRINT or PRINTCONT command in this manner, assign **CHAR** type to its parameter and specify the message to be printed in the *Title* column of the *Procedure Parameters* form.

Alternatively, you can use the MESSAGE and WRNMSG commands to display a message; in the latter case, the user can opt to halt execution of the procedure. The message number must be stored in an **INT** parameter (either via the *Value* column or defined in an earlier procedure step); its content is recorded in the *Procedure Messages* form.

Processed Reports

One of the most common uses of a procedure is to generate a *processed report* — a report whose data undergo processing prior to output. Such output may take the form of a standard report or a table.

A procedure which generates a processed report is considered a special type of procedure. Thus, you have to specify an *R* in the *Rep/Wizard/Dashboard* column of the *Procedure Generator* form. Whenever a procedure of this type is activated, ***it will act like a regular report***. Indeed, there is no way for the user to distinguish between a regular report and a processed one; they look exactly the same.

A procedure of this type is generally made up of the following steps: user input (by means of a parameter input screen or through a form), data manipulation (SQLI step) and running the report with processed data (the report step itself, of type *R*).

A report step has several parameters:

- One parameter for each of the linked files that are created during user input. These parameters are all of **FILE** type, and each is assigned the name of the table to which it is linked as its *Value*.
- Input parameters that are passed on from the procedure to the report query. These, of course, are of the same parameter type they were assigned during the input step.

Unlike other steps containing more than one parameter, there is no need to designate parameter position for a report.

Changing the Report Title

The report title can be changed at runtime in the SQLI step preceding the report step, using the :HTMLFNCTITLE variable.

Example: See step 20 of the **FRTRANS** procedure.

Note: This feature can also be used in HTML documents (see Chapter 6).

Defining Dynamic Column Titles

Report column titles can be defined dynamically. To do so, assign the column title in an SQLI step preceding the report step.

Example: The following changes the title of column #30, so that it displays the title from the procedure rather than what is defined in the report:

```
:COLTITLES = 1;
SELECT ENTMESSAGE('$', 'P', 10) INTO :TITLE1 FROM DUMMY;
:REPCOLTITLE.30 = :TITLE1;
```

Defining Dynamic Report Conditions

Processed reports that are generated by a procedure can also include dynamic conditions that are defined while running the report. To use this option, add a local variable called REPCONDITION (**FILE** type) to an SQLI step preceding the report step, as follows:

```
SELECT SQL.TMPFILE INTO :REPCONDITION FROM DUMMY;
```

You can now define additional report conditions via queries whose output is written to the REPCONDITION variable (in ASCII format).

Example: See step 40 of the **WWWDB_PORDERS_A** procedure.

Running a Sub-Procedure

A procedure can be included as part of another procedure. This is useful, for instance, when you wish to use the same set of steps within several related procedures, or when you wish to repeat the same set of steps several times within a single procedure.

►To include one procedure within another, use the *Procedure Link* form, a sub-level of the *Procedure Generator*.

Running a Procedure

A procedure may be run in one of three ways:

- directly by means of a program
- via the menu to which the procedure is attached
- via the form to which it is linked.

Using a Program to Run the Procedure

You can use the *Run Procedure* program (accessed from the *Procedures* menu) or the *Run Report/Procedure* program (run by Direct Activation from the *Procedure Generator* form) to execute a procedure. This method is particularly useful when testing a newly developed procedure for bugs.

Activation from a Menu

In order for a procedure to be activated from a menu, it must be linked to that menu. To create that linkage, take the following steps:

1. Enter the *Menu/Form Link* form, a sub-level of the *Procedure Generator* form.
2. Indicate the name of the menu in question and specify *M* in the *Type* column.

3. Specify an integer to determine the order of the procedure within the menu.
4. Disregard the *Background Execution* column. This is only relevant for procedures activated from within a form.

Note: The *Menu Items* form, a sub-level of the *Menu Generator*, serves a similar purpose.

Direct Activation from a Form

Direct Activation of a procedure from within a form can take place in either the foreground or the background. If it is in the background, the user will be able to continue work in the form while the procedure is being executed. Indeed, printouts of processed reports and documents are generally run in the background. In contrast, in a procedure executed in the foreground, the user has to wait until it is completed to continue work in the form.

When a procedure is run by Direct Activation, the form record on which the cursor rests is input into the procedure's PAR parameter (thus, you must label the input parameter "PAR"). Additional steps in the procedure can then use this parameter.

Example: In the third SQLI step of the **ADDPARTTREE** procedure, there is a link to the PAR variable (which is defined as an input column in the first SQLI step).

To allow for Direct Activation of a procedure from within a form, take the following steps:

1. Enter the *Menu/Form Link* form, a sub-level of the *Procedure Generator* form.
2. Indicate the name of the form in question and specify *F* in the *Type* column.
3. Specify an integer to determine the order of the procedure within the form's list of Direct Activations.
4. If the procedure is to be executed in the background, flag the *Background Execution* column.

Note: The *Direct Activations* form, which is a sub-level of the *Form Generator* form, serves a similar purpose.

Help Messages

You can use the *Procedure Generator* to create on-line help messages for an entire procedure and/or any parameter appearing in its input screen. Help for the procedure itself is specified in the *Help Text* form, a sub-level of the *Procedure Generator* form; parameter help is designated in the *Help Text* form, a sub-level of the *Procedure Parameters* form. Whenever help messages (for the entire procedure or individual parameters) are added or modified, the date and time of this revision appear in the *Help Date* column of the *Procedure Generator* form.

Line Breaking

Line breaking for help messages is set automatically. Therefore, you can freely edit the text in the *Help Text* form without having to worry about the length of lines. You should therefore refrain from the use of hyphenation. To force the beginning of a line, use the code \n.

Example: The help for the **AGEDEBTCUST2** procedure says:
This report displays the amount owing from individual customers on a daily basis, together with the number of days payment is overdue. \n
\n
The amounts owing are ordered by date, from the oldest to the most recent.

Referring to Other Entities

Often, you will want to refer to other ***Priority*** entities in your help text. As entity titles are easily changed (even by the user), whereas their names are relatively fixed by the time you begin to create help messages, ***Priority*** offers a mechanism for referring to an entity by *name* in the help text; its current *title* will consequently be displayed when the help is called up by a user.

To refer to a specific entity in a help message, designate the entity name followed by a period and its type, where *F*=form, *R*=report and *P*=procedure, enclosing all of this in curved brackets (e.g., {WWWSHOWORDER.P}.

When your help messages are more or less complete, you should check that all designated entity names and types are correct. To do so, run the *Check Entity Names in Msgs/Help* report (*System Management → Dictionaries → Messages and Help Text*). Any mistakes that are uncovered will have to be corrected in the appropriate *Help Text* form.

Rules for Customizing Procedures

- You cannot revise a standard procedure. Instead, you must copy it and make revisions to the copy. If the procedure runs one or more reports, you may need to copy the reports as well (depending on the type of revisions desired; see Chapter 4).
- Include the appropriate four-letter prefix (e.g., **XXXX_WWWSHOWORDER**) in the name of any procedure you create.
- When creating a copy of a standard procedure that runs a program, do not change any of the parameters that are transferred to the program.

Chapter 6: Documents

Introduction

You can use a procedure to generate a document. This kind of procedure collects data from several reports and displays it using an Internet browser. Each report creates a file, and the last step of the procedure combines all these files into one displayed file. The final document can be activated from a form via Direct Activation (retrieving input from the record it was activated from), or it can be run from a menu, in which case the user chooses the relevant records during parameter input. The reports in such a procedure must handle one record from the base table at a time, identified by the autounique key.

Example: In the **WWWSHOWORDER** procedure, each report handles one record from the **ORDERS** table at a time. Each report receives an ORD parameter, which contains the value of the autounique key of the relevant order.

Creating the Input for the Document

The INPUT parameter of the linked file for the procedure's main table must be named PAR. If the procedure is to be activated from the menu, you must define this parameter as input (specify *I* in the *Input* column) and specify the *Column Name* and *Table Name* of the table column that will be linked. If the procedure is to be activated only from a form, it is not necessary to specify the PAR parameter as input, but you do have to indicate column and table name.

Declaring the Cursor

The HTMLCURSOR command is used to create the cursor that goes over the records stored in the linked table. The only thing you should write in this step is the cursor query. The first column in the query should be the autounique key of the table. You can retrieve more columns in the cursor for sort purposes. The link to the table occurs in the background of the HTMLCURSOR step.

Example: In the **WWWSHOWORDER** procedure, the **ORDERS** table is linked to the PAR parameter. The first column in the query is the **ORD** column.

In a non-English system: If the document is in English (i.e., it is assigned the value *E* in the *HTML Document* column of the *Procedure Generator*), you will not be able to pass any procedure parameters from a previous step to a step that comes after the HTMLCURSOR step, as the latter are run in a separate process. You can work around this by writing the parameter values into a temporary table before the HTMLCURSOR step, and then reading these values from the table in a later step.

Going Over the Records

The next step in the procedure is an SQLI step in which you retrieve the records from the cursor. The system automatically opens the cursor that was declared in the HTMLCURSOR step, runs the LOOP and closes the cursor.

The value of the first parameter in the cursor is saved in a variable called :HTMLVALUE. This is a system variable of **CHAR** type, and it must be converted to an integer using the ATOI function. Using :HTMLVALUE you can retrieve the relevant record (the one being printed) from the main table of the document in question.

Example: In the **WWWSHOWORDER** procedure,
SELECT ORDNAME,ORD,ORDSTATUS,BRANCH INTO :PAR1,:$.ORD,:STAT,:BRANCH FROM ORDERS WHERE ORD = ATOI(:HTMLVALUE) AND ORD <> 0;
The :$.ORD variable will be passed to the reports.

Displaying the Document

The last step in the document procedure should be the INPUT command. In the *Procedure Parameters* sub-level form, you must list all the text file parameters that were sent to the reports in the procedure. The INPUT step combines all these files into a single HTML file.

To understand how the text files are positioned on the page, the HTML page can be viewed as a matrix. Each text file is placed in that matrix in the order you specify in the *Proc. Parameter-HTML Design* form, a sub-level of *Procedure Parameters*.

Thus, for each parameter, specify its location in terms of row and column. All the reports defined in row 1 will appear at the top of every new page of the document. For example, you would probably like the company logo to appear at the top of every page. You can also specify the percentage of the width that each report will occupy (in the *Width %* column).

Defining Print Options

You can offer the user a number of print formats to choose from, based on selected reports included in the procedure. First, name the print format in the sub-level form, *Print Formats;* then, specify which reports will be displayed in this format using the *Reports Included in Print Format* form. You can also offer the user the option of printing attachments together with the main document. To do so, include the HTMLEXTFILES command towards the beginning of the procedure, recording a step query for it. This command has no parameters.

Example: See the various print formats defined for the **WWWSHOWORDER** procedure. See also the HTMLEXTFILES command in this procedure.

Executing Reports that will Create the Document

All reports that are part of the document should receive at least two parameters. The first is the value of the autounique key of the record currently being worked on (e.g., the sales order in question). The second parameter should be of **ASCII** type, and must contain "OUTPUT" in the *Value* column. This parameter will appear again in the final INPUT step of the procedure, in which all the text files created by the reports are combined into one document.

Sending Documents by Automatic Mail or Automatic Fax

There are several print/send options to choose from when working with documents. In addition to the standard options, you can use calculated report columns to include an Automatic Mail option (automatically sends the report to the designated contact as an e-mail attachment) and an Automatic Fax option (automatically faxes the report to the designated contact).

To enable the Automatic Mail option, include a report column with the title *#MAIL*. This option opens a customer or vendor task, depending on the contact in the document. If the contact in question is an external contact of another company (i.e., is linked to the customer/vendor in the document via the *External Contacts* form), add a *#CUSTNAME* or *#SUPNAME* report column that stores the number of the customer/vendor in the current document. Otherwise, the task will be opened in the name of the contact's company.

To enable the Automatic Fax option, include a report column with the title *#FAX*. You also have the option of adding a *#DEST* column, which will cause the contents of this field to appear at the top of the fax alongside "**Attn**."

Document Design: Forcing Display of the Line Number

For HTML printouts that require line numbering, you can prevent users from hiding the line number by including a report column whose title is {#LINE}. Such a column will not be displayed in the design utility, will appear untitled in the document and will automatically be assigned the first column position.

Notes:

Only one {#LINE} column can be included in any given report.

Only user designs created in ***Priority*** version 17.3 and higher will be affected.

A Special SQLI Step: Creating E-Documents

You can customize procedures to send digitally signed documents (e-documents) to designated customers via e-mail. In the SQLI step used to go

over the records before printing a document, check whether the following conditions are met:

- :SENDOPTION = 'AMAIL' (the user has selected the Automatic Mail option in the Print/Send Options dialogue box).
- :WANTSEDOCUMENT = 1 (the user has flagged *Are sent e-mails digitally signed by Outlook* in the Mail Options dialogue box).
- In the **CUSTOMERS** table, the value of the **EDOCUMENTS** column for the designated customer is *Y*.
- The record in question has received a final status.
- The document has not yet been printed (the *Printed* column is not flagged for the record in question).

If the above conditions are met:

- add the words " - Digitally Signed" to the document's header.
- flag the document as *Printed.*
- to flag the e-mail for synchronization with ***Priority*** (optional), set the value of the :EDOCUMENT variable to 1.

Example: See the print formats defined for the **WWWSHOWCIV** procedure.

Creating an E-Document using Procedure Code

Assuming that the user running the procedure has been granted the privileges required for digitally signing PDF documents, an HTML document may be converted into a PDF and digitally signed from within the procedure itself. This is done by adding the *–signpdf* option to the WINHTML command.

Example: The following code will create a digitally signed PDF of the sales order in which ORD = 100:

```
:ORD = 100;
:PDFFILE = '../../SOMEFILENAME.pdf';
EXECUTE WINHTML '-d', 'WWWSHOWORDER','','-v', :ORD, '-signpdf', '-pdf', :PDFFILE;
```

Saving a Certified Copy when Printing a Document

You can define a procedure so that, when the document is printed or sent by e-mail (:SENDOPTION = 'PRINT' or 'AMAIL'), a certified copy is saved that can be printed later (using the *Create Certified Copy* program).

In order to save a copy:

- Include an HTMLEXTFILES step which goes over the attached files.
- In the first INPUT step, set the :HTMLPRINTORIG variable to 1.
- In the SQLI step after the HTMLCURSOR step, set the :SAVECOPY variable to 1.

These variables are updated during runtime, so that if several documents are being printed together, the variables will be set for each individual document, depending on its status.

Example: See the **WWWSHOWCIV** procedure.

Note: See also the SHOWCOPY command in Chapter 5.

The Letter Generator

The *Letter Generator* allows users to design letters that can be sent to a third party (e.g., a customer or vendor) using dynamically populated data fields. In the *Letter Generator* form, users can assign the letter a name and choose which procedure is used to create it. This procedure creates letters for a specific recipient or group, using a linked form record (e.g., a record in the *Customers* or *Vendors* form) to populate any data fields.

The *Letter Generator* form has two sub-levels:

- *Remarks* — Used to design and edit the content of the letter (including data fields).
- *Attachments* — Used to attach relevant files to the letter.

Creating a Letter

Procedures used to create and send letters are like other procedures that generate HTML documents, and use a similar structure. A procedure of this type should contain the following steps, in this order:

- INPUT
- HTMLCURSOR
- An SQLI step to retrieve the record for which the letter is being created (by means of the HTMLVALUE variable) and insert values from this record and related records into the **LETTERSTACK** table (see below).
- Any reports that should appear at the top of the letter (e.g., the company logo and document number).
- A MAILMERGE step, which populates any fields in the formatted text with the corresponding values. The MAILMERGE program receives the following parameters:
  - A linked table which receives the formatted text, after all fields have been replaced by the corresponding values.
  - A variable of **INT** type that holds the autounique value of the relevant record.
  - A variable of **CHAR** type that holds the name of the column containing the record's autounique value (e.g., **CUSTOMERS.CUST** or **SUPPLIERS.SUP**).
- The LETTERSTEXT report. Like other reports that are run with processed data, this report receives two parameters:
  - A linked table containing the processed data (in this case, the linked table from the MAILMERGE step, in which all fields have been replaced by the corresponding values).
  - An **ASCII** parameter with the value "OUTPUT", which receives the report's output.

This report can be defined once and then reused as is for any procedures used to create letters, with no need for further modification.

- Any reports that should appear at the bottom of the letter (e.g., the user's signature).
- The INPUT command, which combines all the text file parameters that were sent to the reports in the procedure into a single HTML file.
- The END command (which ends the procedure).
- The source report for the letter. This report lists all data fields that can be included in the letter, and is based on the **LETTERSTACK** table (see below).

Example: See the standard **CUSTLETTER** procedure.

The LETTERSTACK Table

This table is used as a basis for the procedure's source report. The table's unique key contains the following 4 columns:

- **USER**
- **STATUSTYPE**
- **KEY1**
- **KEY2**

During the SQLI step, this table is populated with values taken from the record for which the letter is being created (and related records), as follows:

- **USER —** Receives the name of the current user.
- **STATUSTYPE —** Receives the type of document for which the letter is being created, represented by the BPM system assigned to that document type (see Chapter 13). For example, for letters to customers, this column receives the value *5*.
- **KEY1 —** Receives the autounique value of the record for which the letter is being created (e.g., **CUSTOMERS.CUST**). This is usually the value of the HTMLVALUE variable in the procedure.
- **KEY2 —** This column can receive the autounique value of a related record used in the letter's source report (e.g., for letters to customers, this column can receive the autounique value of the customer's main contact).

Notes:

To offer users the option of sending a letter using the Automatic Mail and Automatic Fax print/send options, the letter's source report must include columns with the revised title *#MAIL* and *#FAX*, respectively.

To offer users the option of sending attachments together with the letter, include the HTMLEXTFILES command after the HTMLCURSOR step, recording a step query for it.

Letters can also be attached to mail messages sent by the BPM mechanism, provided the same document type is assigned to both the letter and the BPM flow chart.

A single letter can be linked to more than one procedure (in the *Letter Generator* form). In such a case, users will be able to choose the desired procedure in the Print/Send Options dialogue box.

Chapter 7: Interfaces

Introduction

This chapter introduces two special interface tools. The form load utility (**INTERFACE** program) serves both to import data directly into a ***Priority*** form (from an external text or XML/JSON file or an internal load table) and to export form data to a file or table. In contrast, the table load utility (**DBLOAD** program) imports data into an interim table from a tab-delimited text file (Excel files can be converted into tab-delimited files using a utility program; see p. 147). SQL statements (a load query) can be recorded for the load and will be executed as it is performed. The table data can then be displayed in a form, in which further manipulations can be made (e.g., by Direct Activation of a procedure), before it is loaded from the interim table into the appropriate ***Priority*** form using the **INTERFACE** program. In fact, these two tools are often used together: the table load to add data to an interim table, and then the form load to load the data into a regular form.

You ***cannot*** change an existing form load or table load. You can, however, make use of existing load tables, interim tables and load procedures. Sometimes it is enough to create your own table load and use existing entities for the rest. See example, as well as tips on finding standard ***Priority*** interfaces, at the end of this chapter.

Important! The records stored in ***Priority*** tables are ***always*** inserted into those tables via ***Priority*** forms. ***Never*** insert records directly into ***Priority*** tables, as this will bypass the integrity checks and other actions defined in the form triggers. Even if you repeat all existing logic of the form in question, your code can cause bugs in the future. For example, if mandatory columns are added to the table in some future software version, with default values that are filled in within the form, this will be ignored by your code!

The Form Load

The form load is used both to import records directly into ***Priority*** forms and to export records from those forms.

Example: You write a procedure that receives the customer name and a list of ordered items as input. The procedure opens a sales order for the customer in input, and inserts the items into the *Order Items* sub-level form. In this case you must use the *Form Load Designer* to insert new records into the *Sales Orders* form and its sub-levels.

In general, a form load interface is characterized by:

- a unique interface name
- a load table in ***Priority*** or an external file to which data is loaded (or from which it is exported)

- definitions linking the table columns, file columns or XML tags to individual form columns
- parameters affecting the load.

These definitions are recorded in the *Form Load Designer* form and its sub-levels (*System Management → Database Interface → Form Load (EDI)*).

Form Load Attributes

►To open a form load and record its attributes, use the appropriate columns in the *Form Load Designer* form.

Load Name and Title

The load name is a short name by which the load is identified by the system. The following restrictions apply:

- Only alphanumeric values (uppercase and lowercase letters and digits) and the underline sign may be used (no spaces).
- The name must begin with a letter.
- You may not use a reserved word (a list of reserved words appears in the *Reserved Words* form — *System Management → Dictionaries*).
- The name assigned to any newly created form load must include a common four-letter prefix (the same one you use for all entities that you add to ***Priority*** for the customer in question; e.g., **XXXX_LOADFNC**).

The load title is the means of identifying the load when executing it from the menu.

Module

Each form load belongs to a given ***Priority*** module. As different modules are included in each type of ***Priority*** package, users are restricted to those form loads whose modules they have purchased. Assign a new form load the "Internal Development" module; this way you (and your customers) will be able to use it no matter which modules of ***Priority*** have been purchased.

Load Parameters

A number of parameters will affect the form load. Some of these can also be designated during execution of the load itself:

- *Do Not Skip Lines* — Flag this column when you want the **INTERFACE** program to continue loading records of the current record type once an error occurs. Leave this column blank to stop the insertion of records of the current record type when an error is encountered.
  Note: The same purpose is served by the *–noskip* parameter, which can be included in the form load execution (see more details below).

Example: The form load is supposed to open two sales orders, each of which contains a few order items. If the **INTERFACE** program encounters an error while loading the first item of the first sales order and the *Do Not Skip Lines* column is ***not*** flagged, then the program will skip to the next order. If the column is flagged, then the program will continue loading the remaining items of the first order.

- *Ignore Warnings* — ***Priority*** forms generate two types of messages: errors and warnings. Leave this column blank if you want the **INTERFACE** program to treat warning messages as errors. To ignore warning messages, flag this column.
  Note: The same purpose is served by the *–w* parameter, which can be included during the form load. Furthermore, you can define a specific run of the form load so that warnings appear in the errors report even when they are "ignored", by means of the *–W* parameter (see more details below).
- *HTML Text* — When exporting ***Priority*** data to an application that supports HTML tags (such as another ***Priority*** application), flag this column. HTML text definitions, such as fonts, sizes and colors, will be transferred intact, together with the text.

Loading from/to a Load Table

Each load can be based on either a load table or an external file. The following explains how to work when using a load table to import or export data.

Briefly, it is recommended that the following steps be taken:

1. Determine which form columns need to be loaded and into which forms. Pay special attention to mandatory columns that are not filled in automatically; these must be included in the load.
2. Determine how the interface will be structured. Choose between using the default **GENERALLOAD** table as your load table and designing your own table.
   Tips:
   - If needed, you can add columns to the **GENERALLOAD** table.
   - If you create your own table, in addition to the mandatory columns (see below), use column names that are relevant to the loaded data, as this will facilitate programming and interface definitions.
3. Enter the needed data into the load table (e.g., build a program).
4. Define the interface in the *Form Load Designer*. You may discover, as you define it, that you need to make adjustments to the load table. If you decide to switch load tables (e.g., create your own table), you will have to redefine the interface.
5. Run the **INTERFACE** program.

The Load Table

►Designate the name of the table in the *Load Table* column of the *Form Load Designer* form, indicating any *Record Size*.

The load table must contain certain columns, defined as follows (name, type, width, title):

- **LINE** (**INT**,8,'Ln')
- **RECORDTYPE** (**CHAR**,3,'Record Type')
- **LOADED** (**CHAR**,1,'Loaded?')
- **KEY1** (**CHAR**,20,'Key 1')
- **KEY2** (**CHAR**,20,'Key 2')
- **KEY3** (**CHAR**,20,'Key 3')

The **LINE** column must be the unique key of the table.

Notes:

The predefined **GENERALLOAD** table meets all these criteria and also includes several columns of each type.

The predefined **GENERALLOAD_T** also meets these criteria. Moreover, it contains an extra TITLE column, which can be used to store a message that will be added to the form's error message (if there is one). See, e.g., the **DELWTASKITEMS** procedure.

In some cases (see below) you may need more than three key columns. As you define the forms included in the load, the *Form Load Designer* will warn you if there are not enough keys in the designated table. If you need more keys, add them to the relevant **GENERALLOAD** table or design your own table.

Once a record is successfully loaded into a form (after the **INTERFACE** program is run), the **LOADED** column is assigned the value *Y* and the values of the key columns are updated by the new record that was opened. If there is an autounique key, that will serve as **KEY1**. If not, the columns making up the unique key will be inserted in as many keys as necessary. Therefore, you need at least as many key columns in your table as the number of columns in the unique key.

Example: If the **INTERFACE** program opens new customers, in every record that was successfully loaded into the **CUSTOMERS** form, the value in the **KEY1** column will equal the value in the **CUST** column.

Forms in the Load

Use the *Forms to be Loaded* sub-level form to specify which forms participate in the load. These usually consist of a root form (e.g., **ORDERS**) and one or more sub-levels (e.g., **ORDERITEMS**).

Indicate the name of each form, as well as the code representing its record type. Each level in the form tree must be assigned its own unique record type (e.g., **ORDERS** = 1; **ORDERITEMS** = 2).

Tip: After recording a form, run the *List of Sub-level Forms* report by Direct Activation to view all its sub-levels (one level down only).

For each form, if you want the load to overwrite existing records in the sub-levels, flag the *Replace Form Data* column. Leave this column blank to add the new records to existing ones.

Notes:

Use of this flag to overwrite form data is based on the assumption that the existing record will be deleted successfully. If the form in question has a PRE-DELETE trigger, it is not advisable to flag this column, as the trigger may interfere with deletion.

This flag is used mainly for text forms.

Link Form Columns to Table Columns

Use the next sub-level, *Link Form Cols to Load Tbl Cols*, to indicate which form columns are equivalent to which load table columns and to specify the order of column insertion.

While the Choose list of form columns only includes updatable columns in the form, the **INTERFACE** program can also insert values into hidden columns. This is useful, for instance, when you want to create a form load that ***updates*** records. Due so with caution, however. In most other cases, it is not good practice to insert values into hidden or read-only columns, as this may contradict some checks defined in the form.

Example: To update the unit price of ordered parts whose records already exist in the **ORDERITEMS** form, you would define one of the load table columns as updating the hidden **ORDI** column in that form. In this case, make this column first in order of insertion. But to specify the ordered part, use the **PARTNAME** column. ***Do not*** use the hidden **PART** column, even though this may seem easier, as the CHECK-FIELD trigger of the **PARTNAME** column will be bypassed and this is likely to have an adverse effect on record insertion.

If you want the **INTERFACE** program to treat empty strings and zero values as a true value, flag the *Insert Null Values* column.

Example: Flag this column to load records into the **ORDERITEMS** form for ordered parts with a unit price of 0. Otherwise, the **INTERFACE** program will ignore the 0 value in that column and insert the default unit price of the item instead by activating the form's trigger.

Default Values

Use the next sub-level, *Default Value for Column*, to assign a default value to be loaded into the form column. If the load table column is empty, the value specified in this sub-level form will be loaded into the form table. When the form load interface is used to export data, if the form column is empty, the default value will be exported.

Adding Line Items to an Existing Document

When an interface that adds line items to a document is executed, by default the new items are inserted first in the document (that is, they receive a smaller line number than existing records). For instance, if a given interface adds lines to an existing order that contains two lines, the new record will appear on line 1, the first existing order item will move to line 2 and the second existing order item will move to line 3.

If you want to change the position of the new record (e.g., move it directly after the first line item), do the following:

1. Define the interface so that existing lines are retrieved (e.g., by linking the **INT1** column in the load table to the **KLINE** or **ORDI** column).
2. Add a column to the load table to hold the internal ID of the first line (**KLINE** or **ORDI**).
3. In the load table, insert the record of the new line to be added after the record that retrieves the first line.

Loading from/to a File

As mentioned, each form load can be based on either a load table or a file. The following explains how to work when using an external file to import or export data. Before you begin defining the interface, make sure you know the structure of the file you will be using.

Tip: If you want to create a new form load based on an existing one, retrieve the desired form load and run the *File Definitions for Form Load* report by Direct Activation from the *Form Load Designer* form.

Defining the File

►Designate the name of the file, its record size and its file type in the appropriate columns of the *Form Load Designer* form.

If the file is stored in the *system\load* directory, you can specify just the filename. Otherwise, give the full path to the file.

You can use one of three types of files for the form load:

- an ASCII or Unicode text file with fixed column widths
- an ASCII or Unicode text file with data separated by tabs
- an XML/JSON file (using either ASCII or Unicode character encoding).

Note: ***Priority***'s interface tools can import data from either ASCII or Unicode (UTF-16) files, and will automatically recognize the format used. Data exported from ***Priority*** for use in outgoing interfaces will be saved in ASCII format, unless otherwise specified.

Forms in the Load

Use the *Forms to be Loaded* sub-level form to specify which forms participate in the load. These usually consist of a root form (e.g., **ORDERS**) and one or more sub-levels (e.g., **ORDERITEMS**).

Indicate the name of each form, as well as the code representing its record type. Each level in the form tree must be assigned its own unique record type (e.g., **ORDERS** = 1; **ORDERITEMS** = 2). This must match the record type in the file itself.

Notes:

Record type is irrelevant for XML/JSON files. As the column is mandatory, record any value here.

After recording a form, run the *List of Sub-level Forms* report by Direct Activation to view all its sub-levels (one level down only).

For each form, if you want the load to overwrite existing records in the sub-levels, flag the *Replace Form Data* column. Leave this column blank to add the new records to existing ones. This is only relevant, of course, when the root record (e.g., an order) already exists. If the load is used only for inserting new root records, leave the column blank to make the load faster.

Link Form Columns to Fields in File

Use the next sub-level, *Position of Form Columns in File*, to match the data in the file to specific form columns. For text files, also indicate the order of insertion. In a file with tab separators, identify the file data by its column number. Otherwise, indicate the number of the first and last characters that determine the field (in XML/JSON files, this determines the order of insertion).

If you want the **INTERFACE** program to treat empty strings and zero values as a true value, flag the *Insert Null Values* column (see example above). If the file contains **REAL** values (or shifted integers) without a decimal point, indicate the number of *Digits After Decimal*.

For text files, indicate where the record type is located in the file. To do so, use the *Position of Record Type in File* form (a sub-level of the *Form Load Designer*). Specify the column number (in a tab-separated file) or the position of the first and last character that defines this field (in a fixed-width file).

Default Values

Use the next sub-level, *Default Value for Column*, to assign a default value to be loaded into the form column. If the specified position in the file is empty, the value specified in this sub-level form will be loaded into the form table.

When the form load interface is used to export data, if the form column is empty, the default value will be exported.

Additional Definitions for XML/JSON Files

If you are loading to or from an XML/JSON file, you must define the tags. Specifically, you need to take the following steps:

1. From the *Form Load Designer* form, run the *Prepare XML Tags by File Defs* program by Direct Activation. The structure of the file appearing in the *File Name* column will be analyzed and transferred to the *XML Tags for Interface* sub-level form. This file should be saved in the *system**load*\company directory, where *company* is the name of the current company (SQL.ENV).
   Note: If you are exporting data to an XML file, use a sample file with the desired tags to create the structure.
2. Enter the *XML Tags for Interface* form and check results. For each tag, the data in the first record appears in the *Value* column. If you want this (or any other value) to be used in all records, regardless of definitions in the file, specify *C* in the *Type of Value* column. If necessary, revise the value.
3. If you are exporting data to an XML file and there are attributes for any tags, record them in the *XML Attributes* column.

Example: For an XML tag defined as <custname type='string'>1001</custname> you would record "type = 'string' " in this column.

---

4. Return to the *Forms to be Loaded* form and its sub-level *Position of Column in File*. Use the next sub-level, *Definition of XML Tags*, to link each form column to the appropriate tag. If the tag is a date, you can also define the *Date Format*, indicating how the date value will be displayed. You can use any of the available SQL date formats, such as MMDDYY or MM/DD/YY (see Chapter 2).

Note: Once you have created a form load design that uses an XML file, ***Priority*** automatically enables users to export data from the main form of this load design to an XML file. In this case the *XML File* option in the *Mail* menu of the relevant form will be enabled (in Windows). When it is selected, the user gets a choice of interfaces to run. The system indicates where the output file has been saved.

Additional Definitions for Exporting Data

When you are exporting data from ***Priority*** forms to a file, you can use the *Outgoing Interface Definitions* form (a sub-level of *Position of Form Columns in File*) to define the following:

- *Align* – determines how to align the columns in the file (left or right); useful for number columns.

- *Date Format* – determines how date values will be displayed. Again, you can use any of the available SQL date formats, such as MMDDYY or MM/DD/YY.
- *Padding w/Zeroes* – useful for number columns.

Executing the Form Load

There are a number of alternate ways to execute a form load.

- Run the *Load Data* program from the *Form Load (EDI)* menu. Indicate whether to import data (*Load*), load it a second time (*Reload*), or export data (*Upload*).
- Activate the load from an SQLI step in a procedure or from a form trigger. To execute the load in this way, use the following syntax (parameters are explained in the next section):

  EXECUTE INTERFACE '*interface_name',* '*msgfile*', ['-L', '*link_file*'], ['-stackerr', '*stackerr_file'*], ['-w'], ['-ns'], ['-nl'],
  ['-nv'], ['-noskip'], ['-enforcebpm'], ['-t'], ['-W'], [-m] ['-o' **|** '-ou' [, '-f', '*output_file*']], ['-debug', '*debug_file*'], ['-repeat'], ['-I', '*table_name1*', '*link_file1*' [, …'-I', *table_name10*', '*link_file10*']], '-v';

- Include the form load as an interface step (type *I*) in a procedure. You can use the same parameters as in the previous option, except that they are listed in the *Procedure Parameters* form (sub-level form of *Procedure Steps*).
- Include the **INTERFACE** program as a step in a procedure. The first parameter must be the name of the form load. Thereafter, you can use the same parameters as specified above.

Note: The last three methods offer more options than the first.

- In addition, users can export data to an XML file from within a form, by using the *XML File* command in the *Mail* menu (see explanation above).

Form Load Parameters

- *'interface_name'* — The *Load Name* used to identify the form load in the *Form Load Designer*.
- '*msgfile*' — The file in which error messages will be recorded. This can later be used to display the resulting message to the user (e.g., "5 out of 6 records were loaded.").
- '-L', '*link_file*' — Use this option when you want the **INTERFACE** program to refer to a linked file of the load table. '-L' tells the load program to use a linked file, and *link_filename* is the file that will be used to link the load table.
- '-stackerr', '*stackerr_file*' – Use this option if you want to have **INTERFACE** program error messages sent to a linked file of the **STACK_ERR** table. '-stackerr' tells the program to use a linked file of the **STACK_ERR** table, rather than the **ERRMSGS** table, and *stackerr_file* is the name of this linked file.

Note: When using this option, error messages will not be sent to the **ERRMSGS** table; thus, the **INTERFACEERR** report will not retrieve any values.

- '-w' — Use this option to have the **INTERFACE** program ignore warning messages (equivalent to the functionality of the *Ignore Warnings* column in the *Form Load Designer*).
- '-ns' — By default, the **INTERFACE** program displays a progress bar as the load is executed. Use this option to disable this utility.
- '-nl' — If the load program encounters errors, it generates an errors report. By default, each line in that report indicates the line in the load table that generated the error. Use this option if you don't want to display the line number in the message.
- '-nv' — When a value generates an error from a CHECK-FIELD trigger, the errors report, by default, displays the name of the column and value that generated the error. Use this option to hide this information.

Example: If you try to open a sales order for customer C000981 and that customer does not exist in the database, the error message in the report would be:

"Line X - Customer Number C000981: Specified item not in database."

If you use the '-nv' option, the message will display:

"Line X - Specified item not in database."

- '-noskip' — Equivalent to the functionality of the *Do Not Skip Lines* column in the *Form Load Designer* (see above).
- '-enforcebpm' — Many forms use business rules (defined in the Business Rules Generator or the Data Generator), BPM rules and definitions of paths between statuses to manage business processes. As part of this process, the Business Rules Generator/BPM utility sends mail to designated users and/or external addresses. You should include this parameter when you want the **INTERFACE** program to apply any rules created in the Business Rules Generator or the Data Generator and to run the BPM mechanism while loading the form data. If you do not include it, mail will be sent, but all business rules and BPM rules will be bypassed and defined paths will be ignored. This means, for example, that custom error/warning messages defined in the Business Rules Generator will be ignored, and that any status can be changed to any status (e.g., a customer shipment recorded against an order will change that order's status to **Closed**, even though this status change is prevented by BPM rules).
  Note: For details on the Business Rules Generator, the Data Generator and BPM, see the *User Interface Guide*.
- '-t' — When no file type is defined in the *Form Load Designer*, use this parameter to indicate that the file being loaded has tab separators. This option is useful when you do not know the file type in advance. In such a case, you must define form column positions in both ways (for both fixed-width and tab-separated files) and then include this parameter when the load is of a file with tab separators.
- '-W' — Use this option to display warning messages in the *Load Errors* report even if you defined the form load to ignore such messages.

- '-m' — Use this option to break up error messages into several lines (see example in Chapter 15).
- '-o' — Use this option when exporting data. When exporting to a file, the data will be written to the file recorded in the *Form Load Designer* using ASCII character encoding. When loading to a table, the program will insert records in the defined load table.
  Note: This option is often used together with '-f'.
- '-ou' — This is similar to the '-o' option, except that data will be written to the file using Unicode (UTF-16) character encoding.
  Note: This option is often used together with '-f'.
- '-f', '*output_file*' — When exporting data to a file (using the '-o' or '-ou' option), include this parameter to write the output to a different file than the one recorded in the *Form Load Designer*.
  Note: This is necessary when you have recorded a sample file (used to define XML tags) in the *Form Load Designer*.
- '-debug', '*debug_file*' — Parameters used in debug mode; the **INTERFACE** program will write all operations executed by the form load into the specified debug file. This is similar to the '-g' option in other ***Priority*** tools (see Chapter 8).
- '-repeat' — Equivalent to the *Reload* option (when the program is run from the menu). Use this option to reload lines that were not successfully loaded in a previous run (see more below).
- '-l' (lowercase "L"), '*table_name*', '*link_file*' — Use this option when you want the **INTERFACE** program to refer to a linked copy of the designated table. This is similar to the '-L' option, except that you can specify any table (not necessarily the defined load table). This is useful when you want to export data to a table rather than a file. You can then use the linked copy of the table to store the results.
- '-v' — Use this option if you want the **INTERFACE** program to check the structure of the input file. If the type of a value (e.g., **INT, REAL, DATE**) is incompatible with the expected type for that position, an error is generated and the file will not be loaded.

Dealing With Errors and Reloading

If errors were encountered by the **INTERFACE** program, they can be found in the *Load Errors* report (**INTERFACEERR**). These errors are saved in the **ERRMSGS** table, which consists of the columns: **LINE**, **TYPE**, **MESSAGE** and **USER**. The unique key of this table is **USER**,**TYPE**,**LINE**. That is because this table stores messages from many types of system programs, and the messages are unique to each user. That is, two different users can run the **INTERFACE** program, and each will see only messages from his/her own load. The type for errors generated by the **INTERFACE** program is *i*.

There are various methods of displaying error messages to your own customers. Here are a number of suggestions:

- In the error message you create, refer explicitly to the errors report, using its entity name {INTERFACEERR.R}. The entity title will then appear to the user with a link to the report. All the user needs to do is click on the link.

- Include the **INTERFACERR** report as a procedure step, using GOTO to skip this step if there are no errors.
- Link the errors report to the same menu as the procedure that runs the load.
- In a procedure query, define a MSG parameter of **ASCII** type. Then use
  SELECT MESSAGE FROM ERRMSGS WHERE TYPE = 'i' AND USER = SQL.USER ASCII :$.MSG;
  and pass the MSG parameter to a PRINT procedure step.
- If you know there will be no more than one error message, you can use
  SELECT MESSAGE INTO :PAR1 FROM ERRMSGS WHERE TYPE = 'i' AND USER = SQL.USER AND LINE = 1;
  and then use ERRMSG to display the message.

If some of the records were not successfully loaded, you may execute the same load again, with the same data, using the *Reload* or '–repeat' option. The **INTERFACE** program will then run only on those records for which the **LOADED** column is ***not*** assigned the value *Y*.

Example: The load should open orders and insert items into the order line. If one item failed to be loaded (e.g., the *Part Number* was not found in the *Part Catalogue*), you can repair the load table (define the part or fix the number) and re-execute the load, and the item will be inserted into the order that was already opened.

Note: Once the form load is successful, you may wish to export it for installation on a different server (e.g., if you are running the **INTERFACE** program on a test installation and you wish to copy it to the production server). To do so, return to the *Forms to be Loaded* form and run the *Upload Interface* program from the list of Direct Activations.

Executing a Form Load from a Form Trigger or Step Query

Programs that prepare privileges for procedures and forms also check the SQL statements in all form triggers and procedure steps. If any interface is executed from a form trigger or procedure step, the required privileges are prepared whenever the trigger or procedure in question is activated. For example, when using the above syntax (EXECUTE INTERFACE '*interface_name',*), the program prepares privileges for the '*interface_name'* interface for the user in question.

Sometimes, however, you may wish to define an SQL variable that refers to the interface to be executed (e.g., :MYINTERFACE = 'SOMEINTERFACE';), rather than specifying the desired interface directly. In such a case, the privileges program will not be able to identify the interface being executed. If the user has privileges for the form or procedure itself, but not for the interface ('SOMEINTERFACE'), they will not receive any errors, but the form or procedure may not function correctly.

It is therefore recommended that you refer to any relevant interfaces from within the form/procedure in such a way that they are identifiable by the privileges program, while ensuring that they're not actually executed at that point. For example, include each interface in a separate EXECUTE command,

but run it from within a GOTO command that jumps to step 999, or add the interfaces as additional procedure steps, but place them after an END command. If the variable can refer to more than one interface, make sure to include a reference to each possible interface using one of these methods. In this way, the privileges program will be able to identify any interface that may be executed from the form or procedure and will prepare all required privileges.

Example: See the BUF7 trigger in the **DOCPACK** form.

Deleting Records from a Form

The form load can also be used to delete records from a form. In fact, the very same interface definition can be used both to insert/update form records and to delete them, the sole difference being the definition of the record type in the load table or file.

Note: If you use the form load to delete records, make sure that you don't flag the *Replace Form Data* column in the *Forms to be Loaded* form, as this will cause the form load to fail.

►To delete records from a form, record @ before the number value assigned to the record type.

Example: See the POST-FORM2 trigger in the **ORDERITEMS** form, which deletes irrelevant lines from the form.

Table Load

Introduction

A second interface tool allows you to import data into an interim table in ***Priority*** from tab-delimited text files (Excel files can be converted into tab-delimited files using a utility program; see p. 147). During the execution of this table load, you can perform additional processing defined in a set of SQL statements (a load query). Results of the load can be viewed — and revised — in a form based on the interim table. Finally, a procedure can be used to transfer the final data to a regular ***Priority*** form (e.g., **ORDERS**), using the form load defined above.

In general, a table load interface is characterized by:

- a unique load file name
- a specified table or a load query which defines how the load is performed
- input fields defined as variables, which can be included in the load query
- parameters affecting the load.

These definitions are recorded in the *Characteristics for Download* form and its sub-levels (*System Management → Database Interface → Table Load (Interfaces)*).

Table Load Attributes

►To open a table load and record its attributes, use the appropriate columns in the *Characteristics for Download* form.

Defining the Load File

Any load files, which must use ASCII or Unicode encoding, should be stored in the *system\load* directory or one of its sub-directories (specifically, the sub-directory for a particular ***Priority*** company). Its file name must match the name of the load file defined in the *Characteristics for Download* form.

Note: *Priority*'s interface tools can import data from either ASCII or Unicode (UTF-16) files, and will automatically recognize the format used. Data exported from ***Priority*** for use in outgoing interfaces will be saved in ASCII format, unless otherwise specified.

This name is subject to the following restrictions:

- It may consist of up to 20 characters
- Only alphanumeric values (uppercase and lowercase letters and digits) and the underline sign may be used (no spaces).
- The name must begin with a letter.
- You may not use a reserved word.
- The name assigned to any newly created form load should include a common four-letter prefix (the same one you use for all entities that you add to ***Priority*** for the customer in question; e.g., **XXXX_LOADFNC2**).

If the file in question is stored in a company sub-directory, also flag the *Sub-directory* column (and make sure you run the load from the correct company). If data is separated by tabs, flag the *Tab Separator* column. You may also specify a brief description of the load file in the *Description of Data* column.

Defining the Load

You have two options in defining the load:

- You can designate the load table and have ***Priority*** create an automatic query, or
- You can record the load query manually.

In both cases you must also define the input file. This consists of all table columns (in the case of an automatic query), or the variables that are input in each field (in the case of a manual query).

The automatic query inserts all data in the input file into the columns of the specified table. Of course, the input file must contain all unique key columns in the target table (with identical names) in order for the load to succeed. It need not contain the autounique key; if there is no value for this key, it will be assigned automatically during the insert.

A manual query should be constructed if you want to add processing during the table load. This is a set of SQL statements which determine how the load

is to be executed. The statements that make up the load query are executed upon each line of the input file, one at a time.

Automatic Load Query

To make use of the automatic load query, designate the name of the table in the *Table for Auto Load* column of the *Characteristics of Download* form. Then define the input file in the sub-level form, *Input Record Fields*. In the *Variable* column, record the name of each table column into which data should be inserted. Indicate its type and position (first and last character). You can also add a description in the *Title* column.

Note: In a file with tab separators, column width is meaningless.

Manual Load Query

If you want to record a query manually, first define the input file in the *Input Record Fields* sub-level form. On each line, record a variable that is input into a field, to be used in the load query. For each variable, designate its type and define its position within the file. You can also add a description (in the *Title* column).

Note: In a file with tab separators, variable width is meaningless.

Next, record the SQL statements making up the query in the *Load Query* sub-level form. You can include ERRMSG and WRNMSG commands. An error message (ERRMSG) will cause the load of a given input line to fail. The error message is then written in the **ERRMSGS** table. The failure to load one input line has no effect on the loading of another. A warning message is also written to the **ERRMSGS** table, but does not cause the input line to fail. Like in forms and procedures, you can use message parameters :PAR1, :PAR2 and :PAR3. The messages themselves should be recorded in the *Error & Warning Messages* form. SQL statements are stored in the **LOADTEXT** table; messages are stored in the **LOADMSG** table.

Tips:

To check the SQL statements in your load query for syntax errors, prior to activation of the load, run the *Check Syntax* program by Direct Activation from within the *Characteristics for Download* form.

You can track changes to load queries once they have been included in prepared version revisions. For details, see Chapter 9.

Loading the File

There are several ways to execute a table load:

- Run the *Download a File* program from the *Table Load (Interfaces)* menu.
- Activate the load from an SQLI step of a procedure, using the following syntax (parameters are explained in the next section):

EXECUTE DBLOAD ‘-L’, ‘loadname’, ['-I', 'input_file'], ['-I'], ['-T', 'table', 'linkfile'], ['-g', 'debug_file'], ['-ns'], ['-N'], ['-E', 'unloaded_file'], ['-M'], ['-C'], ['-B'], ['-U'], ['-u'], ['-v'], ['msgfile'];

- Include the table load as a load step (type L) in a procedure, indicating the file name as the first parameter. You can use the same parameters as in the previous option, except that they are listed in the *Procedure Parameters* form (sub-level form of *Procedure Steps*).

Note: The last two methods offer more options than the first.

Table Load Parameters

- '*loadname*' — The *File Name* used to identify the table load in the *Characteristics for Download* form (also the name of the load file).
- ‘-I’ (uppercase "i") — Use this option when you want the **DBLOAD** program to input data from the *file.in* file (stored in the *system\load* directory). Alternatively, use the ['-i', '*input_file'*] option to specify a different input file.
- ‘-T’, *table, linkfile* — Use this option when you want the **DBLOAD** program to load data to a linked copy of the designated table.
- '-g', '*debug_file*' — This option creates a file that displays each query and its execution. This is similar to other ***Priority*** debugging tools that offer the –g option (see Chapter 8).
- ‘-ns’ — By default, the **INTERFACE** program displays a progress bar as the load is executed. Use this option to disable this utility.
- '-N' — Use this option if you do not want the **INTERFACE** program to clear the **ERRMSGS** table when executing a load. Instead, the program will append the contents of the newest unloaded file and message file to the **ERRMSGS** table (see below).
- '-E', '*unloaded_file*' — Use these parameters to create a file of all lines from the input file that failed to be loaded. This file will be stored in the *system\load* directory, unless another path is specified. It can then serve as an input file in its own right, once the problem that caused the load to fail has been solved.
- ‘-M’ — Use this option to have the **INTERFACE** program rename the input file to *file.bak* .
- ‘-C’ — If the load table contains numeric values formatted as strings, use this option to have the **INTERFACE** program remove commas from any numbers that appear in strings.
- ‘-B’ — Use this option to have the **INTERFACE** program remove any text enclosed in square brackets.
- ‘-U’ — If the load table contains a **USERS** column, use this option to have the **INTERFACE** program ignore data in this column (data are inserted with **USER** = *0*).
- ‘-u’ — Use this option if you want the **INTERFACE** program to specify the current user when messages are inserted into the **ERRMSGS** table. Otherwise, messages are inserted with **USER** = *0*.
- '-v' – Use this option if you want the **DBLOAD** program to check the structure of the input file. If the type of a value (e.g., **INT**, **REAL**, **DATE**) is

incompatible with the expected type for that position, an error is generated and the file will not be loaded.

- '*msgfile*' — The name of the file in which to hold messages. This file will be stored in the directory from which the **DBLOAD** was activated (*system**load* or *system**load*\company), unless another path is specified. It contains the following:
  - A message indicating how many lines of the input file have been successfully loaded.
  - In the case of a fatal error (the structure of the input file was not defined, there was no load query or there was a syntax error in one of the SQL statements), a message explaining why the load failed altogether.

Viewing Load Messages

Any messages (error/warning) generated by the **DBLOAD** program will appear in the *Download Messages* report (**DBLOADERRS**). These messages are stored in the **ERRMSGS** table with the value *L* in the **TYPE** column, and **USER** = *0* (or **USER** = SQL.USER if you use the '-u' option. In every execution of a load, the messages from the previous execution are deleted (unless you use the -N option).

Converting an Excel File to a Tab-delimited Text File for DBLOAD

Use the **EXL2TXT** command (from a trigger or Step Query of an SQLI step) to convert an .xlsx file to the tab-delimited text file required for table loads.

Example:

```
EXECUTE WINAPP 'p:\bin.95', '-w', 'EXL2TXT.exe', :F, :T;
```

(where **p:** represents the full path to *bin.95*)

Combining Table Loads with Form Loads

You will often find it useful to combine table loads with form loads. That is, you create a **DBLOAD** that loads data into an interim table, and then display such data via a form based on that interim table. Users can use the interim table form to check the loaded data and to fix records if necessary. You can also create a procedure to manipulate the data (e.g., to add a *C* to all customer numbers) and allow users to run it by Direct Activation from the form.

When satisfied with results, a procedure can be run to execute your **INTERFACE** program load. After the form load is completed, each record that was loaded successfully is flagged, those that failed to be loaded can be fixed, and the load program can be executed again with the '-repeat' option.

Example: The **LOADDOCUMENTS_C** procedure (*Load Counts into Interim Table*) loads a file into the **LOADDOCUMENTS_C** form (*Interim Table - Inventory Counts*). Then, the **LOADDOCUMENTS_C2** procedure (*Load Counts from Interim Table*) opens an *Inventory Count* document in the **DOCUMENTS_C** form. Note that the **LOADDOCUMENTS_C2** procedure executes the load by running the **INTERFACE** program, whose first input parameter is the name of the form load.

Tips for Finding Existing Interfaces

You may find it useful to peruse (and possibly use) various components of the standard interfaces provided with ***Priority***. The following helps you to locate these interfaces.

Interfaces for a Specific Form

One helpful tool is the *Form Interfaces* (**FORMINTERFACES**) form, which is a sub-level of the *Form Generator*. This form displays any interfaces in which the form participates. This is not only useful when looking for interfaces, but also important information when you want to revise the form in a way that impacts the interface (e.g., add a mandatory column).

Interfaces for a Specific Form Column

You can also view a list of all interfaces built for a given form column in the *Interfaces for Column* (**FCLMNINTER**) form, which is a sub-level of the *Form Columns* form (itself a sub-level of the *Form Generator*). This is not only useful when looking for interfaces, but also important information when you want to revise the form column in a way that impacts the interface (e.g., change its width).

Existing INTERFACE and DBLOAD Programs

There are a number of ways to find available form loads and table loads, both standard and customized:

- Run the *Procedure Steps* (**PROGREP**) report (*System Management → Generators → Procedures → Procedure Reports*). In the input screen, designate steps of type I (for form loads) or type L (for table loads).
- Form load only: Run the same report for procedures that have the **INTERFACE** program as a step.
- Run the *SQL Development* (**WINDBI**) program (*System Management → Generators → Procedures*) to find the query that runs the form load or table load. From the **Queries** menu, select **Find String** and then record 'EXECUTE INTERFACE' or 'EXECUTE DBLOAD' in the input screen.

Interfaces in General

There are a number of interface menus (e.g., for inventory counts, for sales invoices) that include a table load procedure, an interim table form and a form load procedure. You may find that the interim table form and form load procedure already meet your needs (e.g., if you want to load an inventory count file of data collected from a peripheral device), so that you only have to create a **DBLOAD** program. To find such interface components, retrieve by LOAD* in the various generators (for forms, procedures and/or menus).

Chapter 8: Debug Tools

Introduction

Priority gives you the option of executing entities (forms, procedures and interfaces) using debug mode. When you execute the entity in debug mode, every SQL query is recorded in a file you specify. This is very useful for debugging forms, procedures and interfaces.

You can also debug simple reports, i.e., reports run from the menu or by Direct Activation (as opposed to processed reports, which are executed by a procedure — in which case you debug the procedure). Finally, you need to optimize forms, reports and SQL queries.

Debugging a Form, Procedure or Interface

You debug a form or procedure by running it via a special ***Priority*** tool. From the *Tools* top menu in the Windows interface or the *Run* menu in the web interface, select *Run Entity (Advanced)*.

- To run a form in debug mode in the Windows interface:

  WINFORM *FORM_NAME* -g *debug_file*

- To run a form in debug mode in the web interface:

  FORM_NAME -g *debug_file*

Example:
WINFORM ORDERS -g ..\..\orders.dbg /* in Windows */
ORDERS -g c:\tmp\orders.dbg /* in web interface */

- To run a procedure in debug mode in the Windows interface:

  WINPROC -P *PROCNAME* -g *debug_file*

- To run a procedure in debug mode in the web interface:

  PROCNAME.P -g *debug_file*

Example:
WINPROC -P WWWSHOWORDER -g ..\..\wwwshoword.dbg /* in Windows */
WWWSHOWORDER.P -g c:\tmp\wwwshoword.dbg /* in web interface */

Note: You can also run a procedure in debug mode from within the *Procedure Generator* by Direct Activation.

- To run a ***Priority Lite*** procedure in debug mode (in the Windows interface only):

  WINHTMLH *PROCNAME* -g *debug_file*

Example:
WINHTMLH WWORDERS -g ..\..\wwworders.dbg

To execute a form load in debug mode, you need to include two additional parameters when running the **INTERFACE** program: the first must be '-debug' and the second must be the name of the debug file. The program then records all queries that were executed into the debug file. For further details, see Chapter 7.

Example: EXECUTE INTERFACE LOADORDERS 'c:\tmp\msg', '-debug', 'c:\tmp\dbg';

To execute debugging a table load in debug mode, you need to include two additional parameters when running the **DBLOAD** program: '-g', as well as the name of the debug file. Again, the program records all executed queries into the designated file. For more details, see Chapter 7.

Debugging a Simple Report

To debug a simple report (one ***not*** run from a procedure), dump the report's query using the *SQL Development* (**WINDBI**) program (*System Management* → *Generators* → *Procedures*). From the **Dump** menu, select **Report** and then record the internal name of the report in question. Results, for example, look like this:

```
/*
*
* Report ORDBYPROJMANAGER : Orders per Project Manager
*
*/
/*  Orders per Project Manager  */

SELECT USERS.USERLOGIN AS 'Project Manager',
CPROFTYPES.TYPECODE AS 'Type of Sale (Code)',
CPROFTYPES.TYPEDES AS 'Type of Sale-Descrip',
ORDERS.CURDATE AS 'Order Date', ORDERS.ORDNAME AS
'Order', DOCPROJ.PROJDES AS 'Project',
ORDSTATUS.ORDSTATUSDES AS 'Order Status',
ORDERS.DETAILS AS 'Details', ORDERS.DISPRICE AS 'Total Price
w/o Tax', CURRENCIES.CODE AS 'Curr'

FROM DOCUMENTS, CURRENCIES, ORDSTATUS, DOCPROJ,
ORDERS, CPROFTYPES, USERS

WHERE (ORDSTATUS.MANAGERREPOUT <> 'Y')
AND (ORDERS.ORDSTATUS = ORDSTATUS.ORDSTATUS)
AND (ORDERS.ORDTYPE = CPROFTYPES.CPROFTYPE)
AND (ORDERS.PROJ = DOCUMENTS.DOC)
AND (ORDERS.CURRENCY = CURRENCIES.CURRENCY)
AND (DOCPROJ.MUSER = USERS.USER)
AND (DOCUMENTS.DOC = DOCPROJ.DOC)
AND (1=1)

ORDER BY1 ASC, 2 ASC, 4 ASC, 5 ASC;
```

Note: You can also run a report in debug mode from within the *Report Generator* by Direct Activation.

Optimization

Whenever you customize a ***Priority*** entity, it is important to optimize that entity:

- After adding columns to an existing form, you must check the optimization of the form.
- After customizing a simple report or creating a new one, you should check the report.
- When writing any SQL query (inside a form trigger, procedure step or load query), you should check optimization of the query you wrote.

►To optimize, run the *SQL Development* (**WINDBI**) program. From the **Optimization** menu, select the appropriate option.

Note: For SQL queries, you can also select **+ optimizer** from the **Execute** menu. Or you can select **+ execution** (from the same menu), which executes the query and show the steps of execution; for each step, it also indicates how many records where retrieved.

Table Access

There are three types of table access:

- **Direct** — One specific record is retrieved.
- **Skip** — Some records are retrieved according to keys.
- **Sequential** — All records are retrieved.

Generally speaking, in form optimization, only one table should have a sequential access (the form's base table). All other tables should have direct access.

Example: If you check the optimization of the **ORDERS** form, only the **ORDERS** table should have sequential access. The rest should have direct access.

If the form is a sub-level (e.g., **ORDERITEMS**), the form's base table should have skip access. In this context, if a table has a key with more than one column (e.g., {ORD, KLINE} in the **ORDERITEMS** table), a condition on the first column of the key will create skip access. Optimization of the entity will determine how quickly it will be executed (how quickly records are going to be displayed in a form or how long execution of a report or procedure will take).

Note that the query is actually passed to the database engine (SQL), which determines the actual order of tables. Nonetheless, the optimization can help you to easily detect a query that misses some joins or a table that lacks some keys. You should try to avoid sequential access to the tables, as that usually means the query will be slow.

Advanced Debugging

In addition to the above debugging options available from within ***Priority***, you may occasionally wish to inspect the queries being sent to the database itself (SQL Server or Oracle).

If no profiler is installed, you can open ***Priority*** in trace mode, so that all queries sent to the database are written to one or more trace files.

To open ***Priority*** in this mode, you must first set the directory in which trace files will be saved (e.g., *C:\tmp*). To do so, open a command line and execute one of the following commands:

- When working with a **SQL Server** installation:

  SET SSDEBUG=C:\tmp

- When working with an **Oracle** installation:

  SET ORADEBUG=C:\tmp

Note: You must specify an existing folder.

Next, open ***Priority*** from the command line. The easiest way to do this is to record the target path of your desktop shortcut for ***Priority*** (e.g., *D:\priority\bin.95\WINMENU.exe*).

All queries that are sent to the database while working in this environment will be written to trace files, which are saved in the specified folder.

Logging

The journal function provides a way to record messages of a given severity level to the server log. It is used from within an SQLI step in a procedure.

A [Log] section in the *tabula.ini* file determines which messages are logged (based on the message severity).

Message Severity Levels

The following is the list of message severities, in ascending order (1 being the lowest severity, 6 being the highest).

JOURNAL_DEBUG	1
JOURNAL_TRACE	2
JOURNAL_INFO	3
JOURNAL_WARNING	4
JOURNAL_ERROR	5
JOURNAL_FATAL	6

Usage in *Priority* Procedures

EXECUTE JOURNALP *'level'*, *'message'*;

Example: The following code will cause a message of a given severity level to be written to the server log.

```
:MSG = "Statement failed to execute. Please help.";
/*message to be written to server log */

:SEV = 4;
/*message severity level */

[Some group of SQL statements that will fail]
/* on failure */

EXECUTE JOURNALP :SEV, :MSG;
```

Tabulai.ini Definitions

In order to control which messages will be recorded to the server log, define the following [Log] section in the *tabula.ini* file:

[Log]

Server Path=*'path_to_server_log'* (e.g., S:\Priority\log)

Server Level=*'minimum_level'* (e.g., 4)

Client Path=*'path_to_client_log'* (e.g., C:\tmp\priority\log)

Client Level=*'minimum_level'* (e.g., 4)

Note: All messages (both automatic and manual) having a severity level greater than or equal to the '*minimum_level'* will be recorded in the log. Specifying 0 is equivalent to 5; that is, messages with levels 5 or 6 will be recorded in the log. To record messages of any severity, specify a '*minimum_level'* of 1.

Chapter 9: Installing Your Customizations

Working with Version Revisions

In order to properly handle your customizations, you should work with two ***Priority*** installations — one in which you do the programming and another in which you run tests. The method for handling this is as follows: After creating your customizations in the programming installation, create an upgrade file and install it in the test installation. Only once you are satisfied with results should you install it on the production server.

As revisions are maintained per user, it is imperative for all programmers to work in their own usernames while performing the programming.

Note: In order to execute a DBI operation – i.e., anything that affects a table, table column or key – you must belong to the privilege group of the superuser (*tabula*) and the PRIVUSERS system constant must be set to 1.

Version Revisions

Version revisions are a built-in tool for moving customizations from one ***Priority*** installation to another. ***Priority*** automatically keeps track of any modifications you make to any entity. All you need to do is group these revisions together and prepare a shell file using standard forms and programs.

Steps for Creating Version Revisions

The basic steps needed to create a version revision are:

1. Enter the *Version Revisions* form (*System Management* → *Revisions*). Record a short description of the customization in question and fill in the mandatory columns in this form. A number will be assigned to the revision automatically.
2. Enter the sub-level form, *Revision Steps*. A detailed list of modifications appears by code. Flag whichever modifications you wish to include in the shell file. They should all be related to the customization in question. The order in which you link these lines determines their order in the upgrade file.
   Note: The lines in the *Revision Steps* form are recorded in the name of the user who made the modification. This way, if you have more than one programmer, each can track his/her own changes.
3. Once you have linked all the relevant modifications, create the shell file by running the *Prepare Upgrade* program by Direct Activation from the *Version Revisions* form. The shell file will be called *NN*.sh (where NN is the number assigned to the revision) and stored in the *system\upgrades* directory.

Explanation of the Modification Codes

DBI	Update of the database (tables, table columns, keys).
DELDIRECTACT	Deletion of a Direct Activation.
DELFORMCOL	Deletion of a form column.
DELFORMLINK	Deletion of the link between a form and its sub-level.
DELMENULINK	Deletion of the link between a menu and its menu item.
DELPROCMSG	Deletion of a procedure message.
DELPROCSTEP	Deletion of a procedure step.
DELREPCOL	Deletion of a report column.
DELTRIG	Deletion of a form trigger.
DELTRIGMSG	Deletion of a trigger message.
TAKEDIRECTACT	Linkage of a Direct Activation to a form.
TAKEENTHEADER	Revision to the attributes of an entity (form, report, menu, procedure, interface), such as its title; in the case of a form, also revision to its screen-painted version.
TAKEFORMCOL	Any type of revision to a form column (e.g., title, sorting, joins).
TAKEFORMLINK	Linkage of a form to its sub-level.
TAKEMENULINK	Linkage of a menu item to its menu.
TAKEPROCMSG	Addition/revision of a procedure message.
TAKEPROCSTEP	Addition/revision of any part of a procedure step (e.g., parameters, step queries).
TAKEREPCOL	Any type of revision to a report column (e.g., title, sorting, grouping).
TAKESINGLEENT	Addition/revision of an entire entity.
TAKETRIG	Addition/revision of a form trigger.
TAKETRIGMSG	Addition/revision of a trigger message.
TAKEHELP	Addition/revision of online help for the designated entity.

Tips for Working with Revisions

- Do not create a version revision until you have finished programming. This ensures that all revisions are numbered in the correct sequence (i.e., when multiple programmers are working in parallel).
- Complete modifications on an entity before linking to the TAKESINGLEENT line. This is because, if you create a new entity, you will find a relevant record with the TAKESINGLEENT code in the *Revision Steps* form. Once you link this record to the version revision, any additional modification to that entity will receive a separate record with a separate code.

  Example: You create a new form and link the relevant modification to a TAKESINGLEENT line. If you then continue to add columns to the form, you will receive additional TAKEFORMCOL lines for each new column.

  However, you should not wait more than a working day. When programming takes more than one day, try to prepare the upgrade at the end of each day. This way, at the end of the programming, you will not have to deal with a very large number of records in the *Revision Steps* form.

- There are also situations in which the order of the lines in the revision is important. Here are some examples:
  - If you create a new document, the reports included in the document must appear before the procedure itself.
  - When a procedure step includes a new form interface, the interface must be included in the revision before the procedure.
  - When you add a new column to a table and then use this column in a form, you must first flag the DBI operation and only then flag the TAKEFORMCOL line.
- Do not prepare the same upgrade twice. If a version revision needs to be modified after the upgrade has been prepared, create a new version revision with your modifications and run the *Prepare Upgrade* program for the new revision.

Tracking Changes to Queries

Several form and reports allow you to keep track of changes in queries appearing in form triggers, SQLI procedure steps and load definitions, once they have been included in a prepared version revision.

Note: Query changes are maintained per version revision. After making your changes, open and prepare a new version revision. If you reprepare an existing one, the previous change will not be saved, as the new query text will overwrite the old one.

To view previous versions of a column trigger, a row or form trigger, an SQLI step or a load definition:

1. Enter the relevant form (*Form Column Triggers*, *Row & Form Triggers*, *Procedure Steps* or *Characteristics for Download*, respectively) and retrieve the appropriate record.
2. Enter the *Previous Versions* sub-level form. This displays details of all version revisions that include the current query: the date of the revision, its number, a short description, the version number and the signature of the programmer.

Note: Only revisions created after ***Priority*** version 17.3 is installed will appear.

---

3. Enter the next sub-level form, *Previous Versions – Text*, to view the version of the query.

To view differences between the selected version of the query and other versions:

1. Return to the *Previous Versions* form and select *Track Changes* from the list of Direct Activations.
2. In the input screen, under *Text to Compare*, choose between the *Current Version* (the latest version in effect) and the *Previous Version* (the one immediately prior to the selected revision). As is common practice, additions are marked in blue; deletions are marked in red strikethrough.

Installing the Language Dictionaries

When users at a customer site use a language other than English, you will need to install the revisions in more than one language. This requires, for instance, that any revision is translated to the desired language in the language dictionaries provided by ***Priority*** (*System Management → Dictionaries → Translation*).

Preparing Upgrades for Other Languages

1. ***Before you even begin programming*** for this customer, enter the *System Constants* form (*System Management → System Maintenance → Constant Forms*) and change the value of the UPGTITLES constant to 0. Consequently, no titles (in any language) will be stored in the upgrade file; rather, they will be inserted into a second file (based on the upgrade file) via another program.
2. After every run of the *Prepare Upgrades* program, run the *SQL Development* program and record the following query:

   EXECUTE INSTITLE *fromfile, tofile*;

   where *fromfile* is the upgrade file you have just created (i.e., *..*\s*ystem**upgrades*\NN.*sh*, where NN is the version number), and *tofile* is the output file that will include the titles in all languages (the base language of English and any languages for which there are translations in the dictionaries).

   Important! When preparing an upgrade for a system that has a different base language than your own system (e.g., yours is a Hebrew installation, while the customer's is an English installation), you will want the titles to be taken from the translation (*System Management → Dictionaries → Translation*). In this case, record the following query instead:

   EXECUTE INSTITLE '-l', '*langcode*', *fromfile, tofile*;

   where *langcode* is the code of the language from which to take the titles (e.g., '3' for American English), *fromfile* is the upgrade file you have just created, and *tofile* is the output file that will include the titles in the target language.
3. From the **Execute** menu, select **SQLI Interpreter**.

Examples:

For version number 34, the following command would be recorded:
EXECUTE INSTITLE '..\..\system\upgrades\34.sh','..\..\system\upgrades\34-inst.sh ';
The new file in system\upgrades, 34-inst.sh, will contain the titles of the upgrade you created.

For the same version, when upgrading from your own Hebrew installation to the customer's English installation, the following command would be recorded:
EXECUTE INSTITLE -l', '3', '..\..\system\upgrades\34.sh','..\..\system\upgrades\34-inst.sh ';

Modifying DBI Operations in the Revision

If you began programming for the customer without first updating the UPGTITLES constant, you will have to modify all DBI operations in the revision as follows:

Creating a new table

Instead of:

```
CREATE TABLE PRIV_NEWTABLE 'New Table Title' 1
COL1(CHAR,3,'Column 1')
COL2(CHAR,32,'Column 2)
UNIQUE(COL1);
```

Use the following:

```
CREATE TABLE PRIV_NEWTABLE '[Ktitle : PRIV_NEWTABLE]' 1
COL1(CHAR,3,'[Column: PRIV_NEWTABLE/COL1]')
COL2(CHAR,32,'[Column: PRIV_NEWTABLE/COL2]')
UNIQUE(COL1);
```

Creating a new table column

Instead of:

```
FOR TABLE MYTABLE INSERT MYNEWCOL
(INT,13,'My New Column Title');
```

Use the following:

```
FOR TABLE MYTABLE INSERT MYNEWCOL
(INT,13,'[Column: MYTABLE/MYNEWCOL]');
```

Changing a table title

Instead of:

```
FOR TABLE MYTABLE CHANGE TITLE TO 'New Title';
```

Use the following:

```
FOR TABLE MYTABLE CHANGE TITLE TO '[Ktitle : MYTABLE]';
```

Changing a table column title

Instead of:

```
FOR TABLE MYTABLE COLUMN MYCOL CHANGE TITLE TO 'New
Title';
```

Use the following:

```
FOR TABLE MYTABLE COLUMN MYCOL CHANGE TITLE TO
[Column: MYTABLE /MYCOL];
```

When you finish modifying all DBI steps for the revision in question, continue with steps 2 and 3 above (see **Preparing Upgrades for Other Languages**).

Chapter 10:

Creating and Modifying User Report Generators

Introduction

The purpose of a user report generator is to enable users to build their own reports. There are many predefined user report generators in the system, such as the *Customer Data Report Generator* and the *Customer Invoices Report Generator*.

If you have created a customized module in ***Priority***, you may find it useful to design a new report generator. Moreover, if you have added columns to a form that already has a user report generator (e.g., *Sales Invoices*), you may want to add the new columns to the existing generator. In both cases, as you cannot revise a standard report generator, you have to make copies of all the component entities.

Components of the Report Generator

Whether you are creating your own report generator or modifying a standard one, you have to create three new ***Priority*** entities:

- A base report, whose columns form the basis of the user report (defined in the *Report Generator*).
- A form with which the report is constructed (defined in the *Form Generator*); this form retrieves the records from the generator's base report.
- A procedure that runs the user-defined report (defined in the *Procedure Generator*).

Creating Your Own Report Generator

Constructing the Base Report

For your own report generator, you need to construct a new report in the usual way (see Chapter 4). However, in order for it to serve as a base report, the following rules must be maintained as well:

- Any column that the user can choose to include in the generated report must have a revised title.
- Any column that will be used for grouping must have a value in the *Group by* column and an *R* in the *Group Func.* column.
- For any column that might be used for input, flag the *Input* column.

Note: Remember to check all joins, as well as report optimization.

Constructing the Form

To create the form, you first copy the standard **ASSETREP** form and then make certain adjustments to the copy.

Warning! This is the only circumstance in which it is permissible to copy a standard ***Priority*** form. ***Copying a form for any other purpose will have adverse effects.***

---

1. To copy the standard **ASSETREP** form to a new form, from the *Tools* top menu in the Windows interface or the *Run* menu in the web interface, select *Run Entity (Advanced…)* and write the command:
   WINPROC -P COPYFORM
   This will open a new input parameter window, where you need to record the internal name of the form you want to copy (**ASSETREP**), as well as the name and title to be assigned to the new form.

Note: If you wish to create a customized user report generator from a standard report generator form other than **ASSETREP**, make sure to choose one in which the expression defined for the **TYPE** column is *'r'* (in the *Form Column Extension* sub-level of the *Form Generator*). While different values in the **TYPE** column were once used to distinguish between different types of user generated reports, newer report generators all use the same **TYPE** (*'r'*), and assign each report name a different prefix instead.

---

2. The PRE-FORM trigger in the copied form should look like this:
   ```
   :REPEXEC = 0;
   SELECT EXEC INTO :REPEXEC FROM EXEC
   WHERE ENAME = ' ASSETREP' AND TYPE = 'R';
   :PREFIX = 'AST';
   :KEYSTROKES = '%{Exit}';
   ```
   Make the following changes:
   (1) Change **ASSETREP** to the name of the base report you created earlier.
   (2) Change the value of :PREFIX to another three-letter string (e.g., :PREFIX = 'PST';).
3. In the *Form Columns* sub-level form, move to the **ENAME** column. Then enter the *Form Column Extension* sub-level form and change the expression from LIKE 'AST%' to match the string you used in the previous step (LIKE 'PST%').
4. Fix the POST-FIELD triggers for the **TITLE** and **ENAME** columns in the same manner. That is, where ENAME LIKE 'AST%' appears, revise this to ENAME LIKE 'PST%'.
5. Link the **GREPCLMNS** form as a sub-level of your new form.

Constructing the Procedure That Runs the Report

1. Copy the standard **RUNCUSTREP** procedure to a new procedure.
2. In your new procedure, revise the SQLI query in step 10:
   (1) Change the value of the :TYPE variable from *g* to *r*.
   (2) Change the value of the :PAT variable to match the prefix in your new form (e.g., :PAT = 'PST';).

3. If the base report you constructed is to receive input parameters from the procedure (e.g., a date range, a flag to display open orders only), define these parameters in an INPUT step and in the RUNREPORT step. (See, e.g., the FDT and TDT parameters in the **RUNPROJREP** procedure.)
4. Change the name of the report in the last procedure step to match the new base report you created.

Allowing User Access to the Report Generator

- Link the new form and procedure to the relevant menu.

Adding New Columns to a Standard Report Generator

Any revision to a standard report generator – even if all you want to do is to add new columns – requires you to create copies of all component entities and make revisions to the copies.

The difference between modifying a report generator in this way and creating a completely new generator lies in the standard entities that you copy from. Rather than using a new base report, the **ASSETREP** form and the **RUNCUSTREP** procedure as your sources, you should copy from the standard base report that you are revising and its accompany form and procedure.

1. Copy the standard base report to which you want to add columns, creating your own report in which you will make all needed customizations (e.g., copy **INVOICEREP** to **XXXX_INVOICEREP**).
2. Copy the standard form which shares the same name as the standard base report (e.g., copy **INVOICEREP** to **XXXX_INVOICEREP**). Revise the new form as described above with respect to the **ASSETREP** form (see Constructing the Form, above).
3. Copy the relevant **RUN*REP** procedure of the report you copied (e.g., copy **RUNINVOICEREP** to **XXXX_RUNINVOICEREP**), revising the appropriate SQLI query line to :TYPE = 'r'; :PAT = 'XXX'; (where XXX is the prefix you assigned in the form) and changing the name of the report in the last procedure step. This is similar to the case when you construct a new report generator (see Constructing the Procedure That Runs the Report, above).
4. Link the new form and procedure to the relevant menu.

Chapter 11: Creating and Modifying BI Reports

To create a new BI report or modify an existing one, you have to copy (and then revise) several standard procedures and reports:

- procedures that prepare the data for the BI
- the BI report itself
- the procedure that runs the BI report.

You should use as your point of departure an existing BI report that yields data similar to the results you wish to obtain.

The following illustrates the process using the *Order Analysis (BI)* report and its accompanying procedures, using the PRIV prefix to identify the company in which the customization is made.

This should be the same prefix you use for all entities that you add to ***Priority*** for the customer in question.

Procedures that Prepare the Data

1. Copy the **PREPORDERSCUBE** and **PREPORDERSCUBE1** procedures and create two customized procedures: **PRIV_PREPORDERSCUBE** and **PRIV_PREPORDERSCUBE1**.
2. In the **PRIV_PREPORDERSCUBE** procedure, change the procedure in step 20 from **PREPORDERSCUBE1** to **PRIV_PREPORDERSCUBE1**.
3. In the *Step Query* form for the **PRIV_PREPORDERSCUBE1** procedure, make the following revisions:
    a. Change the **TYPE** variable from 'ORDERS' to 'PRIV_ORDERS'.
    Note: As the length of the **TYPE** column in the **EISCUBES** table is 12, assign a value to the **TYPE** variable that is less than 12 characters.
    b. Change the query that inserts the records into the **EISCUBES** table.
    c. Find the line :ENAME = 'EISORDERSREP' (near the bottom) and change the value of the **ENAME** variable to the name of the customized report (**PRIV_EISORDERSREP**).

The BI Report

1. Copy the **EISORDERSREP** report and create the customized **PRIV_EISORDERSREP** report.
2. In the *Report Column Extension* sub-level form, change the value of the **TYPE** column from: = 'ORDERS' to = 'PRIV_ORDERS' (i.e., the value of the **TYPE** variable in **PRIV_PREPORDERSCUBE1**).
3. Drill-down options are created by defining the relevant report columns for input (I). You can do so for any of the following columns:
    - **CUSTNAME**
    - **PARTNAME**
    - **BRANCHNAME**
    - **CTYPECODE**
    - **FAMILYNAME**

- **ACCFAMILYNAME**
- **AGENTCODE**
- **TYPECODE**
- **COUNTRYCODE**

Note: These report columns must have a value in the *Group by* column.

4. In order to provide additional drill-down options for the report, you can redefine any of the above report columns, essentially "substituting" one drill-down option with another. For example, suppose you want to enable users to view sales data for a specific type of sale, broken down by the unit of sale (e.g., hr, kg, ea). Let us further assume that the report in question need not include a drill-down option for the country of sale. In such a case, you can revise the definitions for the **COUNTRYCODE** column, so that the resulting report can display sales data broken down by unit, rather than by country. To do so, revise the value of the *Target Form Name* column in the *Report Column Extension* sub-level of the *Report Columns* form as necessary (e.g., specify **UNIT** as the *Target Form Name*). You can then change the revised titles of any of these columns as necessary (e.g., for the **COUNTRYCODE** column, you can specify the revised title *Part Units*).
5. Modify the rest of the report to suit your needs (e.g., hide unnecessary columns).
   Note: Any column you wish to display in a BI report must meet at least one of the following conditions:
   - The table name and table column defined for the report column also appear in the **EISCUBES** report.
   - The report column is assigned the same column expression as one of the columns in the **EISCUBES** report.
   - The report column is assigned the same title as one of the columns in the **EISCUBES** report.

The Procedure that Runs the Report

1. Copy the **EISORDERSREP** procedure and create the customized **PRIV_EISORDERSREP** procedure.
2. In step 70 (SQLI), change the value of the **ENAME** variable from 'EISORDERSREP' to 'PRIV_EISORDERSREP' (the name of the customized report).
3. Change the report in step 200 from **EISORDERSREP** to **PRIV_EISORDERSREP**.
4. Copy the base pages of the original procedure (the html file in *system\html* and, if needed, the html file in *system\html\lang.*XX, where XX is the language code defined in the *Languages* form), and change the file names to match the new procedure.

Note: After upgrading the system to a new version, if the customized report does not work in the same way as the standard report, it is recommended to copy the base page from the standard BI procedure again.

Chapter 12: HTML Procedures for Priority Lite & Priority Dashboards

Introduction

The HTML procedures used to display reports on the Internet in ***Priority Lite*** or data in ***Priority*** Dashboards are very similar to regular procedures in ***Priority***. However, there are some basic differences, including additional features that support the special needs of these procedures. There are also some new features that allow you to specially design the displayed reports. This chapter focuses on those innovations.

These procedures are displayed on the Internet or in Dashboards by means of HTML pages that are generated by each INPUT step in the procedure. One or more of the following can be displayed in these pages:

- reports
- input parameters from the procedure
- procedure messages (including error messages)
- graphic web parts.

A base HTML page serves as a template for the page that is displayed. This base page defines where to situate the reports, parameters and messages displayed by the procedure. It can then be revised, and pictures, links and text may be added.

Before you can create the base page for each INPUT step, however, you must construct the HTML procedure and its reports.

The Structure of HTML Procedures

There are two essential structural differences between HTML procedures and regular ones:

- Reports can only be displayed in an INPUT step.
- The procedure does not run continuously from beginning to end. Rather, it is interrupted at each INPUT step, at which time it generates an HTML page. It continues where it left off when the user clicks **Go** (or any other link) on that page.

Displaying Reports

1. Generate the report in a REPORT step.
2. Include in that step, in addition to the parameters that are transferred to the report, an additional parameter of **ASCII** type with the value OUTPUT (do not use quotation marks).
3. Use the same parameter in a (later) INPUT step, but this time without any value.

Note: There is no limit to the number of reports that can be displayed in the same INPUT step.

Example: In a procedure that displays the *Sales Orders* form, you can define three reports: one for the header, another for the order items and a third for totals. The procedure in question generates the three reports (in separate REPORT steps) and then displays them together in the same HTML page (i.e., in the same INPUT step).

Retaining Variable Values After the Procedure Stops

As the procedure does not run continuously, but rather stops and starts, its link files and variables "disappear" whenever it is interrupted. To retain the value of a variable:

1. Define it as a procedure parameter.
2. Make sure it is included as a parameter in the INPUT step at which the procedure stops.

The variables in question are recorded in a hidden section of the HTML page, and their values are returned to the procedure when it is run again.

User Identification

Often, there is a need to identify the user who runs the procedure – as a customer, vendor or internal user – and to display relevant user data (e.g., address). User identification is carried out via the company phone book or the list of users (that is, the data in the **PHONEBOOK**, **PHONEBOOKA** and **USERS** tables).

To have the procedure identify the user before it runs:

- In the *Procedure Generator*, choose the appropriate value in the *Internet Access* column. Specify *Y* for all (customers, vendors and internal users); *T* for walk-in customers (e.g., for the Storefront) and *U* for internal users only (e.g., for internal reports, data input by users). Or specify *I* for no privilege checks.

The value of the first parameter in the cursor is saved in a variable called HTMLVALUE. This is a system variable of **CHAR** type, and it must be converted to an integer using the ATOI function. Using HTMLVALUE you can retrieve the relevant record (the one being printed) from the main table of the document in question.

To identify the user from within the procedure, use the SQL.WEBID variable. When users log in using their e-mail address, this variable receives the value of the **PHONEBOOK.PHONE** column. In procedures that are defined for internal users only, users can also log in with their username. In such a case, the SQL.WEBID variable receives the value of the **USERS.USER** column, multiplied by -1.

Tip: To display report data that is specific to a given customer/vendor/user, make the appropriate joins in the report to SQL.WEBID.

Example: To obtain a list of the customer's orders, use the join
WHERE PHONEBOOK.PHONE = SQL.WEBID AND
PHONEBOOK.CUST = ORDERS.CUST

Automated User Identification for Priority Lite Websites

If the procedure can only be run by an internal user (i.e., the value of the *Internet Access* column is *U*), you can also set up automated identification for a particular user via the DOS command prompt.

To do so, first copy the *setuser.exe* application from the *x\priority\bin.95* directory to the local *C* drive on the workstation from which ***Priority Lite*** will be run (where *x* is the network drive on which ***Priority*** is installed).

In a DOS command line, enter the following commands (where *myWebsite.com* is the URL of your ***Priority Lite*** website, *myuser* is the relevant username on ***Priority*** and *mypassword* is that user's password):

set TABULAPORTALURL=*http://myWebsite.com*

c:\setuser *myuser mypwd* Y " 3

c:\setuser *myuser mypwd* Y " 4 */* for clients running Windows Vista and higher*/*

Subsequently, when the user in questions runs a ***Priority Lite*** procedure, he or she should include the following parameter: *&_portal=1*.

Example: To run a procedure called **WWWMYPROC**, enter the URL as follows:
http://myWebsite.com/priority/prihtml.dll?WWWMYPROC&_tabulaini=tabula.ini&_portal=1

If the PC from which this user works keeps track of Cookies, then this login information will be applied whenever he or she enters the web site. Otherwise, the DOS command must be run again each time the user leaves the browser.

Client Identification for Priority Lite Websites

In addition to SQL.WEBID, which receives a value only after the user logs in, another variable of identication – SQL.CLIENTID – is created automatically by the system whenever a new user enters the web site. This variable is of **CHAR** type, with a width of 20.

If the PC from which this user works keeps track of Cookies, then this user will receive the same ID whenever he or she enters the web site. Otherwise, the SQL.CLIENTID is saved only until the user leaves the browser.

Designing HTML Reports for Priority Lite & Priority Dashboards

Any ***Priority*** report can be displayed on the Internet or in ***Priority*** Dashboards, provided that it is run from within an HTML procedure.

Nonetheless, a few forms in the *Report Generator* (see below) enhance its capabilities in two main directions:

- You can specially design the HTML report, so that it has the “look” of a form.
- You can define input columns and links in the report.

The basic idea of specially designed reports is to be able to place the fields that appear in the report wherever you want on the HTML page. That is, you select and place the report fields on the output page (hereafter “form”). To place the fields in the form, you logically divide the form into table cells, and then place each report field in its respective cell. For a detailed explanation, see Chapter 4, Displaying HTML Text in Reports.

Input Columns and Links in the Report

There are several types of input for a report:

- strings, including hidden character input fields (for password input)
- integers
- real numbers
- Choose lists, including radio buttons, check boxes, Multiple Choose lists and Multiple Check boxes.

Three types of links can also be defined:

- to a URL (web link)
- internal link back to the current procedure (the procedure will continue to run from the step where it left off)
- internal link to another procedure (which will be activated by the link).

Input columns and links are defined in the *Link/Input* tab of the *Report Columns–HTML Design* form. Any report column can be made into an input or link column. To choose the type of input or link, use the *Link/Input Type* column.

The number of characters that can be recorded in a given input column is determined either by the width of the report column or by the value of the MAXHTMLWIDTH system constant, whichever is greater. For example, if the width of the report column is defined as 32 and the value of the MAXHTMLWIDTH constant is 20, users will only be able to see 20 characters at a time, but will be able to record up to 32 characters in the column in question. Users can view any data that exceeds 20 characters by scrolling within the input column.

Defining Choose Lists

When an input column is a Choose list, you have to define the query for retrieving the values that will appear in that list. You can do so in one of two ways:

- Via a target form: In the *Report Columns–HTML Design* form, define a target form in the *Target Form (Choose)* column.
- Via a query: Create a trigger and record an SQL query for the list in the *Field Triggers* form, another sub-level of the *Report Columns* form.

The Choose query is very similar to a CHOOSE-FIELD form trigger. You retrieve two or three values, where the first retrieved value is displayed in the Choose list, while the second is returned to the :HTMLFIELD variable in the procedure. The third value (optional) is used solely for sorting purposes. (Of course, you can also sort by each of the other two fields.)

Tip: To make the retrieval conditional upon the values of other fields in the same report, use a variable beginning with # followed by the column number (e.g., :#75 for column 75).

Maintenance of Input and Links

Any HTML page that the user accesses can include a number of reports. Each of these reports may have a number of input columns and links. When the user clicks on one of the links, the procedure runs anew (assuming, of course, that the link entails a return to the procedure). When this happens, the procedure needs to know which link the user has activated and what values he or she specified in the input columns.

In most cases, the report has the format of a table (columns and rows). In order to identify a given field, the system needs to know which column and which row are involved, as well as the value of that field. For this purpose, three variables have been specially created:

- :HTMLACTION – to identify the column (a fixed string)
- :HTMLVALUE – to identify the row (a value returned by one of the report fields other than the current field)
- :HTMLFIELD – to return the value of the current field (relevant for input columns, but not for links).

Note: Of course, for a report that displays only one record, it is sufficient to identify the column (:HTMLACTION).

Example: In the list of parts appearing in the Storefront, each part has at least two links:
(1) a link from the part description (or its picture), which opens a screen displaying more detailed information about the part;
(2) a link which adds the item to the shopping cart.

The procedure needs to receive the correct value for the :HTMLACTION variable in order to know which action to take (e.g., “DETAILS” to display part information or “SHOPPINGCART” to add the item to the cart). In both cases, the value of :HTMLVALUE will be the same (the desired part, i.e., **PART.PART**).

To identify the table column:

- Specify the string which identifies the table column in the *Return Value Name (:HTMLACTION)* column of the *Report Columns–HTML Design* form. This becomes the :HTMLACTION variable in the procedure (see more below).

To identify the table row:

- Specify the number of the column identifying the table row in the *Return Value Column# (:HTMLVALUE)* column. This becomes the :HTMLVALUE variable in the procedure.
  Important! This report column must be either a displayed column or a sort column.

Defining Links in the Report

As explained above, a link in a report can be to a URL or to a procedure (the current one or a new one). To create a web link (URL):

1. Select *W* or *w* in the *Link/Input Type* column.
2. Indicate the number of the column in which the URL is stored in the *Return Value Column# (:HTMLVALUE)* column.

To link back to the same procedure:

1. Select *P*, *p*, *b*, *H* or *N* in the *Link/Input Type* column (depending on the type of window you want to generate).
2. Indicate the columns that determine the return values (**HTMLVALUE** and **HTMLACTION**).

To create a link to a new procedure:

1. Select *P*, *p*, *b*, *H* or *N* in the *Link/Input Type* column.
2. Indicate the columns that determine the return values (**HTMLVALUE** and **HTMLACTION**).
3. Record the column that defines the procedure in the *Internal Link Column #* column. Ensure that the value in this column is the name (**ENAME**) of the procedure.
   Important! This must be a sort column in the report.
4. Return to the *Report Columns* form and move to the column specified in the preceding step.
5. Enter the *Report Column Extension* sub-level form and, in the *Expression/Condition* column, record the name of the procedure you want to activate.

To link to multiple procedures:

- Ensure that the column defining the procedure includes the name of the procedure that needs to be activated for each of the report records (similar to the same process in a form).

Example: In order to display the financial document by its reference number in the journal entry, you need to activate one procedure in the case of an invoice, another procedure in the case of a customer receipt and a third procedure in the case of payment to a vendor.

Handling Input From Report Columns in the Procedure

As explained earlier, three variables help to determine input from report columns:

- :HTMLACTION – a fixed string identifying the column
- :HTMLVALUE – a value returned by a report field (other than the current one) that identifies the row
- :HTMLFIELD – returns the value of the current report field.

When the procedure is run anew (by the Internet user), the :HTMLACTION and :HTMLVALUE variables contain the values from the field that the user clicked to activate the link.

To access the values input by the user:

- Create a cursor-like mechanism that passes through all the report's input columns, using the DISPLAY command within SQL code. Each time the command is activated, the values of a different input column are filled in from the report with the three variables.

Notes:

If several reports are displayed by the INPUT step, the values of input columns will be returned for ***all*** the reports.

If you include a Multiple Choose list or Multiple Check box field in the report, each of the user's selections is stored in a different :HTMLFIELD variable, but only one :HTMLACTION variable is defined (since all values were selected in a single column).

Example:

```
LABEL 1;
DISPLAY;
GOTO 2 WHERE :RETVAL <= 0;
….
LOOP 1;
LABEL 2;
```

Procedures That Work Like Forms (Input Screens)

In addition to procedures that display reports, you can create procedures that enable user input and have the "look" of a form.

Loading Data Into *Priority*

To avoid replication of code (which leads to bugs), load the data into ***Priority*** via an interface to the desired form, using the form load utility (for full details, see Chapter 7).

1. Using **GENERALLOAD** as your load table, create a linked file of all data.
2. Activate the form load interface.
3. Check that all records in the table were loaded.
4. If they were not, display the error message returned by the load program.

Example:

```
LINK GENERALLOAD TO :$.LNK;
/* Insert one or more lines into the linked file */
INSERT INTO GENERALLOAD (LINE,RECORDTYPE,…) VALUES(…);
/* Run the interface on the linked file */
EXECUTE INTERFACE 'MYINTERFACE', SQL.TMPFILE, '-L', :$.LNK;
/* Insert the message that the interface gave to :PAR1 */
SELECT MESSAGE INTO :PAR1 FROM ERRMSGS
WHERE USER = ATOI(RSTRIND(SQL.CLIENTID,1,9))
AND TYPE = 'i' AND LINE = 1;
/* Display the message if any lines were not loaded successfully */
ERRMSG1 WHERE EXISTS (SELECT 'X' FROM GENERALLOAD
WHERE LOADED <> 'Y' AND LINE > 0);
UNLINK GENERALLOAD;
```

User Input Validation and Messages

Input Validation

In many cases, there is a need to validate user input, whether of procedure parameters or of input report columns. To validate user input:

- Add an SQL check (step query) to the INPUT step in the procedure. In the event of an error (i.e., ERRMSG that does not fail), the procedure will display the same HTML page as before, together with the appropriate error message.

Resetting Variables and/or Generating the Reports Again

Sometimes, before the page is re-displayed, you may want to send the procedure back to a step prior to INPUT, so as to reset variables or to generate the same reports anew. To go back to an earlier step:

- Assign an appropriate value, in the INPUT step in question, to the :HTMLGOTO variable ***before*** the code for the error messages.

Note: If no error messages are generated, the procedure ignores the :HTMLGOTO variable and continues with the step that follows INPUT.

Another Way to Display Messages

To display any type of message (including error messages) on the HTML page:

1. Add a MESSAGE step to the procedure, prior to the INPUT step.
2. Record the message number as the value of its (single) parameter.

Adding Explanatory Text

In many procedures, especially those requiring user input, it is helpful to add brief text that guides the user through the actions he or she has to perform.

While this text can be written on the base page itself, it is more efficient to record it as set text directly in ***Priority***. For this purpose, a standard report called **HTMLTEXT** has been created.

To include set text in the procedure:

- Add a REPORT step for **HTMLTEXT** to the procedure (before the INPUT step). Use a single parameter: **OUTPUT** (as in any other report).

To record the set text itself:

1. Enter the *Set Text for Internet Screens* form *(System Management → System Maintenance → Internet Definitions)*. This form displays all Internet procedures, with a line for each REPORT step that runs **HTMLTEXT**.
2. Move to the proper line (find the right procedure title and step number).
3. In the *Set Text* sub-level form, write the explanatory text that will appear on the HTML page.

Input of Text

To enable the user to input text in the HTML screen, use a procedure parameter of **TEXT** type. The user keys in an unlimited number of lines in the text field, and these lines are returned to the procedure via a file linked to the **PROCTABLETEXT** table. For an example, see Chapter 5.

Input of Attachments

The input of attachment files is another special type of procedure input. Use a **CHAR** parameter and specify *Y* in the *Browse Button* column of the *Procedure Parameter Extension* form.

In the HTML screen, the user records the attachment filename. After he or she clicks **Go**, the procedure automatically copies this file to the server in the *system\mail* directory and records the new filename in the procedure parameter. That is, after the INPUT step, the value in the attachment parameter will not be the filename that the user keyed in, but rather the filename in the server.

Defining a Base Page for HTML Pages

As HTML pages are generated by each INPUT step in the procedure, you have to construct ***a separate base page*** for each INPUT step. To create base pages automatically:

- Run the *Create HTML Pg for Step* program, which is a Direct Activation from the *Procedure Steps* form, sub-level of the *Procedure Generator*.

The resultant base page will be in ASCII format and can be found in the *system\html* directory of the ***Priority*** server. Its name is the procedure name followed by the step number.

Example: If you run the program for the **MYPROG** procedure at step 50, you will create a page called MYPROG-50.HTM.

It is advisable to include as little text as possible in base pages, in order to facilitate localization to different languages (not including bi-directional languages such as Hebrew and Arabic) using the same base page.

To this end, it is possible to automatically insert a document's header as the title of the HTML page, using the following syntax:

```
<TITLE><!--| Priority Title |--></TITLE>
```

Similarly, you can insert procedure messages directly into the base page when the document is produced.

Example: To insert message 5 of the **WWWDOCUMENTS_Q** procedure in a base page, add the following HTML tag:
<!--| PRIORITY MESSAGE (WWWDOCUMENTS_Q 5) |-->

Important! Whenever you add a new report or input parameter to a given INPUT step, ***you must re-create the base page*** (i.e., run the program again).

Note: If you are working in Hebrew, the program will prepare the base page in both Hebrew (right-to-left) and English (left-to-right), where the former is stored in the *system\html* directory and the latter is stored in the *system\html\lang3* directory.

Revising the Base Page

Once the base page has been created, you can revise it using any standard editor for HTML page design. You can add pictures, links and text, and you can change the order and position of the reports, parameters and messages that are input into the template.

Writing Dashboard Procedures

The procedures used to display ***Priority*** Dashboards in ***Priority*** and on Outlook (***Priority on Outlook***) are very similar to other HTML procedures. Dashboard procedures, however, are assigned a different value in the *Rep/Wizard/Dashboard* column of the *Procedure Generator*:

There are three types of Dashboard procedures:

- "Basic Dashboards" display live data in a single window without any internal procedures (e.g., **WWWCUSTINFO**). This type of procedure is almost identical to a regular ***Priority Lite*** procedure. However, in the *Rep/Wizard/Dashboard* column of the *Procedure Generator*, make sure to assign the value *D*, indicating that this is a Dashboard procedure.
- "Multi-part Dashboards" consist of a number of smaller Internet procedures, or web parts, arranged within a main window (e.g., **WWWDB_SERVICEMNGR**). When creating internal procedures for this type of Dashboard, assign them the value *d* in the *Rep/Wizard/Dashboard* column of the *Procedure Generator*.
- Like basic dashboards, "Portlets" display live data in a single window without any internal procedures, but on the home page (in the web

interface). In the *Rep/Wizard/Dashboard* column of the *Procedure Generator*, assign them the value *p*.

Creating a New Multi-Part Dashboard

1. Create a copy of an existing procedure (e.g., **WWWDB_SERVICEMNGR**).
2. Revise any HTML pages (stored in the *system\html* directory) in your new procedure as follows:
   Locate the line:
   onload="javascript:DashboardLoad(WWWDB_SERVICEMNGR)
   and replace the procedure name (**WWWDB_SERVICEMNGR**) with the name of the new procedure (e.g., **PRIV_NEWDASHBOARD**).
3. Customize the new procedure (**PRIV_NEWDASHBOARD**) as desired, and attach the desired internal procedures (i.e., procedures with the *Rep/Wizard/Dashboard* value *d*).
   Note: See Chapter 5 for Rules for Customizing Procedures.

Adding a Dashboard Procedure to Outlook

Once you have finished preparing the new Dashboard procedure, you can add it to Outlook from the **Mail** menu in ***Priority*** (**Mail > Mail Options > Outlook > Priority on Outlook**). ***Priority*** Dashboards appear in Outlook under the heading **Priority Dashboards** (in the Shortcuts pane).

CRM Dashboards

A CRM Dashboard is a special type of basic Dashboard procedure, which displays a report of live CRM data. This type of Dashboard procedure is created in much the same way as a basic Dashboard procedure, and should be assigned the value *CRM* in the *Application* column of the *Procedure Generator*. CRM Dashboards that are added to Outlook appear under the heading **Priority CRM**.

Chapter 13: Creating BPM Flow Charts

This chapter explains how to create a BPM status system for a new document. Assuming the new document is called **XXXX_MYDOC**, the following explains how to:

- Create a form called **XXXX_MYDOCSTATS** (and its accompanying table).
- Modify the **XXXX_MYDOC** form to support statuses.
- Create a procedure for the BPM flow chart (and accompanying interfaces).

Notes:

A BPM flow chart can only be created for an upper-level form.

All the names and descriptions of forms, tables, columns, etc. are given here for demonstration purposes and can be changed as desired.

For the purposes of this procedure, the first two keys of the base table of the XXXX_MYDOC form should be:
1. An autounique key
2. A unique key, comprising a single column of **CHAR** or **INT** type.

Creating the Statuses Table

1. Use the *CreateTable* procedure to create a new table with the following attributes:
   Table Name – **XXXX_MYDOCSTATS**
   Table Type – **0** (small table)
2. Define the table's three basic columns:
   MYDOCSTAT – **INT**, 13, *Status (ID)*
   STATDES – **CHAR**, 12, *Status*
   SORT – **INT**, 3, *Display Order*
3. Add a mandatory flag with the following attributes:
   INITSTATFLAG – **CHAR**, 1, *Initial Status*.
4. Add additional flags as you wish (see examples in the standard status forms).
   Note: You do not need to add the flags *Include in ToDo List* or *Inactive Status*, or provide a column for a status description in English (for non-English environments). These columns already exist in the **DOCSTATUSES** table, which will be joined to the **XXXX_MYDOCSTATS** form.
5. Define the following columns as keys:
   AutoUnique – **MYDOCSTAT**
   Unique – **STATDES**

Creating the Statuses Form

1. Use the *Form Generator* to create a new form named **XXXX_MYDOCSTATS**, based on the **XXXX_MYDOCSTATS** table.

2. Define the following outer joins (add a question mark in the *Join ID* column next to the *Join Table* column):
   XXXX_MYDOCSTATS.MYDOCSTAT = DOCSTATUSES.ORIGSTATUSID
   DOCSTATUSES.COLOR = HTMLCOLORS.COLOR
3. Define the following form columns:
   XXXX_MYDOCSTATS.STATDES – *Status*
   XXXX_MYDOCSTATS.INITSTATSFLAG – *Initial Status*
4. Add all of the flags from the **XXXX_MYDOCSTATS** table.
5. Add the following flags from the **DOCSTATUSES** table:
   DOCOPENED – *Include in ToDo List*
   INACTIVE – *Inactive Status*
   ESTATDES (optional for non-English system) – Adds a description in English.
   Note: When recording these columns, do not fill in the *Column Name* or *Table Name* columns; rather, enter the *Form Column Extension* sub-level form and enter the table and column names in the *Expression/Condition* column (i.e., **DOCSTATUSES.DOCOPENED**).
6. Add the following hidden columns (listed are the *Form Column Name*, *Table Name* and *Column Name*, respectively):
   DOCSTATUS, DOCSTATUSES, DOCSTATUS
   SORT, MYDOCSTATUSES, SORT
   STATUSTYPE, DOCSTATUSES, TYPE
   VCOLORNAME, HTMLCOLORS, COLORNAME
   Note: The **STATUSTYPE** column must have an expression 'PRIV_MYBPM', where 'PRIV_MYBPM' is a name for the new BPM system. This name must begin with a four-letter prefix, similar to what you use throughout the system for this customer, and can include up to 10 characters.
7. Move to the line for the **STATDES** column and specify *1* in the *Sort Priority* column.

Form Triggers

The following form triggers need to be created:

- A CHECK-FIELD trigger in the **INITSTATFLAG** column that prevents users from marking more than one status as an initial status:

```
ERRMSG 1 WHERE :$.@ = 'Y' AND EXISTS
(SELECT 'X' FROM XXXX_MYDOCSTATS
WHERE INITSTATFLAG = 'Y' AND MYDOCSTAT <>
:$.MYDOCSTAT);
```

- A PRE-INSERT/PRE-UPDATE trigger that checks the validity of the record. For example, the following checks that the initial status allows document revision:

```
ERRMSG 2 WHERE :$.INITSTATFLAG = 'Y' AND :$.CHANGEFLAG
<> 'Y';
```

- A POST-INSERT/POST-UPDATE trigger that handles record insertion into **DOCSTATUSES**:

```
INSERT INTO DOCSTATUSES(TYPE,ORIGSTATUSID)
VALUES(:$.STATUSTYPE, :$.MYDOCSTAT);
UPDATE DOCSTATUSES SET STATDES = :$.STATDES,
ESTATDES = :$.ESTATDES /* only in non-English system */,
SORT = :$.SORT, COLOR = :$.VCOLOR, INACTIVE = :$.INACTIVE,
DOCOPENED = :$.DOCOPENED
WHERE TYPE = :$.STATUSTYPE AND ORIGSTATUSID =
:$.MYDOCSTAT;
```

Note: If you don't support a non-English environment, don't update the **ESTATDES** column.

- A POST-DELETE trigger, that deletes from **DOCSTATUSES**:

```
DELETE FROM DOCSTATUSES WHERE TYPE = :$.STATUSTYPE
AND ORIGSTATUSID = :$.MYDOCSTAT;
```

- A PRE-FORM trigger that causes all records to be displayed when the form is entered. It should contain the following text:

```
:statustype = 'PRIV_MYBPM'; /* where 'PRIV_MYBPM' = STATUSTYPE
*/
:KEYSTROKES = '*{Exit}';
```

- A POST-FORM trigger with additional checks (e.g., that a status was marked as initial status):

```
ERRMSG 4 WHERE NOT EXISTS
(SELECT 'X' FROM XXXX_MYDOCSTATS WHERE INITSTATFLAG =
'Y');
```

- A PRE-INS-UPD-DEL trigger (or an error message in the PRE-INSERT, PRE-UPDATE and PRE-DELETE triggers) that prevents any manual changes in this form, since all the changes will be made using an interface run by the BPM Chart:

```
ERRMSG 17 WHERE :FORM_INTERFACE <> 1;
```

- A CHOOSE-FIELD trigger that displays the statuses from the **DOCSTATUSES** table:

```
SELECT STATDES, '', ITOA(SORT,3)
FROM DOCSTATUSES
WHERE TYPE = 'PRIV_MYBPM'
AND
#INCLUDE STATUSARCS/EXP1
```

Modifying the New Document

Assigned to

The next step is to add an *Assigned to* column to the new document (**XXXX_MYDOC**). Consequently, when a user changes the status of the document to a status flagged *Include in ToDo List*, a new record will be opened in the *To Do List* form for that user.

1. Use the *Column Generator* to add a new column to the **MYDOC** table: **OWNER** – **INT**, 13, *Owner (ID)*.
2. Add a column to the **MYDOC** form called **OWNER**, with a join to the **USERS** table: MYDOC.OWNER = USERS.USER
3. Add a third column with the following *Form Column Name*, *Table Name* and *Column Name*: **OWNERLOGIN**, **USERS**, **USERLOGIN**. Assign the revised title *Assigned to*. This column should be mandatory (Type *M*); it is filled automatically by the BPM when a status is changed.
4. Add Message 402 to the **XXXX_MYDOC** form. This message should consist of a brief description of the document (e.g., Price Quote, Blanket Sales Order), which will appear as the *Document Type* in the *To Do List* form.

The Status Column

1. Add another column to the **MYDOC** table: **MYDOCSTAT** – **INT**, 13, *Status (ID)*. This column must have the same name as the column of the autounique key in the **XXXX_MYDOCSTATS** table.
2. Add the following columns to the **XXXX_MYDOC** form:
   MYDOCSTAT (joined): XXXX_MYDOC.MYDOCSTAT = XXXX_MYDOCSTATS.MYDOCSTAT
   STATDES (mandatory, not joined): **XXXX_MYDOCSTATS.STATDES**
   STATUSTYPE (calculated, Type **CHAR**): Assign column expression 'PRIV_MYBPM', where 'PRIV_MYBPM' is the STATUSTYPE.
3. Add a POST-FIELD trigger to the **STATDES** column:
   :$.STATUSTYPE = 'PRIV_MYBPM'; {*where* 'PRIV_MYBPM' = STATUSTYPE}
4. Add a POST-FIELD trigger to one of the columns that will fill in the initial status. For example, if you have a **MYDOCNAME** column in **XXXX_MYDOC**, add the following POST-FIELD to that column:
   ```
   SELECT MYDOCSTAT INTO :$.MYDOCSTAT
   FROM XXXX_MYDOCSTATS
   WHERE INITSTATFLAG = 'Y' AND :$.MYDOC = 0;
   ```
5. Add POST-INSERT and POST-UPDATE triggers with the text:
   ```
   GOTO 51 WHERE :$.MYDOCSTAT =:$1.MYDOCSTAT
   :doc = :$.MYDOC;
   :status = :$.MYDOCSTAT;
   :statustype = 'PRIV_MYBPM';
   #INCLUDE STATUSAUTOMAIL/SendStatusMail
   LABEL 51;
   ```
   Note: **MYDOC** should be the *Form Column Name* of the autounique key of the **XXXX_MYDOC** table.
6. If you have an *Allow Revisions* flag for the status, add the **XXXX_MYDOCSTATS.CHANGEFLAG** column to the form, as well as PRE-INSERT, PRE-UPDATE and PRE-DELETE triggers that print an error if the status doesn't allow changes.

Connecting the New Document to the To Do List

1. Add the following hidden column to **XXXX_MYDOC** (listed are the *Form Column Name*, *Table Name* and *Column Name*, respectively): **NSCUST**, **MYDOC**, **MYDOC**. This column contains the autounique key of the **XXXX_MYDOC** table (the same as the **MYDOC** column).
2. Add POST-FIELD triggers, in both the **MYDOC** and **NSCUST** columns, to copy the values from one column to the other. For example, in the **MYDOC**

column, the trigger would be:
:$.NSCUST = :$.@;

3. Add another hidden column to **XXXX_MYDOC**: **STATUSTYPE** (Type **CHAR**, Width 1). Assign the column expression 'PRIV_MYBPM', where 'PRIV_MYBPM' is the STATUSTYPE.
4. If two or more forms share the same base table (e.g., both the **CUSTNOTES** and **CUSTNOTESA** forms are based on the **CUSTNOTES** table), add the following hidden column to those forms that do not appear in the **DOCEXEC** column of the **STATUSTYPES** table: **STATMAILSTATUSTYPE** (Type **CHAR**, Width 1). Assign the column expression 'PRIV_MYBPM', where 'PRIV_MYBPM' is the STATUSTYPE.
5. Link the **DOCTODOLISTLOG** and **DOCTODOLIST** forms as sub-levels of the **XXXX_MYDOC** form.
6. Run the following query to enable activation of the **XXXX_MYDOC** form from the *To Do List*:
INSERT INTO ZOOMCOLUMNS(NAME, TONAME, POS) VALUES('TODOREF', 'MYDOCNAME', X); {*where X = some number*}
7. Every change in the status or user assigned to your document is documented in the **TODOLIST** table. You should also decide whether users will be able to delete records from the document. For example, you cannot delete records in the **ORDERS** form, you can only change their status to one which is flagged as **Canceled**.
If you decide to allow record deletion in your document, you must ensure that any deleted records are deleted from the **TODOLIST** table as well. To do so, add a POST-DELETE trigger to the **XXXX_MYDOC** form:

```
DELETE FROM TODOLIST WHERE TYPE = 'PRIV_MYBPM' AND IV
= :$.MYDOC;
```

Updating the STATUSTYPES Table

The **STATUSTYPES** table is used by the BPM mechanism to recognize the forms and columns involved in the workflow process. It is therefore necessary to find certain values and insert them in this table.

Finding the Values

The following values must be determined:

- **DOCEXEC** – The **EXEC** value of the **MYDOC** form.

```
SELECT EXEC FROM EXEC WHERE ENAME = 'MYDOC'
AND TYPE = 'F' FORMAT;
```

- **STATEXEC** – The **EXEC** value of the **MYDOCSTATS** form.

```
SELECT EXEC FROM EXEC WHERE ENAME =
'XXXX_MYDOCSTATS'
AND TYPE = 'F' FORMAT;
```

- **PROCEXEC** – The **EXEC** value of the procedure used for printing the **XXXX_MYDOC** document. This printout can be attached to a mail message sent by the BPM. The following query assumes this procedure is called **XXXX_WWWSHOWMYDOC**:

```
SELECT EXEC FROM EXEC WHERE ENAME =
'XXXX_WWWSHOWMYDOC'
AND TYPE = 'P' FORMAT;
```

- **STATCNAME** – The name of the column in the **XXXX_MYDOC** form that holds the autounique key of the **XXXX_MYDOCSTATS** table, i.e., **MYDOCSTAT**.
- **DOCCNAME** – The name of the column in the **XXXX_MYDOC** form that holds the autounique key of the **XXXX_MYDOC** table, i.e., **MYDOC**.
- **DOCNOCNAME** – The name of the column in the **XXXX_MYDOC** form that contains the number of the document. This is usually the Unique key of the **XXXX_MYDOC** table (e.g., **MYDOCNAME**).
  Tip: In the standard *Sales Orders* form, it's the **ORDNAME** column; in the standard *Shipment Document* form, it's the **DOCNO** column.
- **OWNERCNAME** – The name of the column in the **XXXX_MYDOC** form that holds the *Assigned to* login, i.e., **OWNERLOGIN**.
- **INITSTATCNAME** – The **INITSTATFLAG** column in the **XXXX_MYDOCSTATS** form.
- **DOCDATENAME** – The name of the column in the **XXXX_MYDOC** form that holds the date the document was recorded (e.g., **CURDATE**).
  Note: This column is used to retrieve the document when performing a full text search; for details, see the **Search** section below.
- **TEXTEXEC** – If the **XXXX_MYDOC** document has a sub-level text form (e.g., the **ORDERSTEXT** sub-level of the **ORDERS** form), the **EXEC** value of this text form.
  The following query assumes this form is called **XXXX_MYDOCTEXT**:

  ```
  SELECT EXEC FROM EXEC WHERE ENAME =
  'XXXX_MYDOCTEXT'
  AND TYPE = 'F' FORMAT;
  ```

- **TEXT2EXEC** – If the **XXXX_MYDOC** document has an additional sub-level text form (e.g., for text in a second language), the **EXEC** value of the second sub-level text form.
  The following query assumes this form is called **XXXX_MYDOCTEXT2**:

  ```
  SELECT EXEC FROM EXEC WHERE ENAME =
  'XXXX_MYDOCTEXT2'
  AND TYPE = 'F' FORMAT;
  ```

Inserting the Data

Run the following query in order to insert the above data into the **STATUSTYPES** table:

INSERT INTO STATUSTYPES(TYPE, DOCEXEC, STATEXEC, PROCEXEC, STATCNAME, DOCCNAME, DOCNOCNAME, OWNERCNAME, INITSTATCNAME, DOCDATENAME, TEXTEXEC, TEXT2EXEC) VALUES('PRIV_MYBPM', DOCEXEC, STATEXEC, PROCEXEC, 'MYDOCSTAT', 'MYDOC', 'MYDOCNAME', 'OWNERLOGIN', 'INITSTATFLAG', 'DOCDATENAME', TEXTEXEC, TEXT2EXEC);
{*where* 'PRIV_MYBPM' *is the* STATUSTYPE*, and* DOCEXEC, STATEXEC, PROCEXEC*,* TEXTEXEC, TEXT2EXEC *are the numeric values from the above queries}*.

Search

Priority's Search tool is used to perform a full text search for documents and attached files in the system. This search mechanism is able to search for specific text in all documents that have a BPM status system (including custom documents), provided these documents are set up as follows:

- In the **DOCDATENAME** column of the **STATUSTYPES** table, specify which column in the document is the date (used to specify a date range for searches; e.g., the **CURDATE** column of the **ORDERS** form).

Creating the Necessary Interfaces

BPM Interface

The next step is to create a form interface to the **XXXX_MYDOCSTATS** form. Changes in the BPM chart will update the form using this interface.

1. Open the *Form Load Designer*.
2. In the *Load Name* column, write the name of the interface: "BPMPRIV_MYBPM", where "PRIV_MYBPM" is the name of the **STATUSTYPE**.
3. In the *Load Table* column, write "GENERALLOAD".
4. In the *Record Size* column, specify "500" (this should be large enough).
5. Enter the *Forms to be Downloaded* sub-level form.
6. In the *Form Name* column, write "XXXX_MYDOCSTATS".
7. In the *Code (Record Type)* column, write "1".

Note: You don't need to fill in the *Link Form Cols to Load Tbl Cols* form (a sub-level of the *Forms to be Downloaded* form), as this form will be filled in automatically when you run the BPM procedure.

Update Status/Assigned User

The **STATUSMAIL** interface is used by the BPM to change the status and assigned user of the **XXXX_MYDOC** form.

1. In the *Form Load Designer*, write the name of the interface in the *Load Name* column: "STATUSMAILPRIV_MYBPM", where "PRIV_MYBPM" is the name of the STATUSTYPE.
2. In the *Load Table* column, write "GENERALLOAD".
3. In the *Record Size* column, specify "500" and flag the *Ignore Warnings* column.

4. Enter the *Forms to be Downloaded* sub-level form.
5. In the *Form Name* column, write "XXXX_MYDOC".
6. In the *Code (Record Type)* column, write "1".
7. Enter the *Link Form Cols to Load Tbl Cols* sub-level form, and create three records, indicating the following *Load Table Column*, *Form Column Name* and *Order*, respectively:
   INT1, **MYDOC**, **1**
   TEXT1, **OWNERLOGIN**, **2**
   TEXT2, **STATDES**, **3**
   where:
   - **MYDOC** is the *Form Column Name* of the column in the **XXXX_MYDOC** form that contains the autounique key of the **XXXX_MYDOC** table.
   - **OWNERLOGIN** is the *Assigned to* column in the **XXXX_MYDOC** form.
   - **STATDES** is the *Form Column Name* of the column in the **XXXX_MYDOC** form that contains the status description.

Creating the Procedure for the BPM Chart

You now need to create a procedure called **XXXX_VISMYDOCSTATS** in the *Procedure Generator*. The procedure steps should be as follows:

- Step 5
  Entity Name: **SQLI**
  Type: *C*
  Step Query (in the sub-level form): #INCLUDE WEBCONST/NotFromJava
- Step 10
  Entity Name: **BPM**
  Type: *C*
  Procedure Parameters (in the sub-level form):
  - Parameter Name: CHR
  - Pos: 10
  - Width: 20
  - Value: PRIV_MYBPM (**STATUSTYPE**)
  - Type: **CHAR**
- Step 90
  Entity Name: **END**
  Type: *B*
- Step 91
  Entity Name: **XXXX_MYDOCSTATS** (Status form)
  Type: *F*

Debugging the BPM

You can run a BPM flow chart in debug mode in the Windows interface. To do so, use the following command:

BPM *StatusType* -g *debugfile*

Example: To debug the *BPM Flow Chart – Sales Orders* (in Windows)*:*
BPM O -g ..\..\bpm_O.dbg

Inserting the Initial Status into the Status Table

The final step is to write a query that will insert the initial status into the **XXXX_MYDOCSTATS** table:

```
:INITSTAT = 'Initial Stat';
:STATUSTYPE = 'PRIV_MYBPM'

INSERT INTO XXXX_MYDOCSTATS
(MYDOCSTAT,STATDES,INITSTATFLAG,CHANGEFLAG)
VALUES(-1,:INITSTAT,'Y','Y');

INSERT INTO DOCSTATUSES (ORIGSTATUSID,TYPE,STATDES,
NEWDOCFLAG)
VALUES(-1, :STATUSTYPE, :INITSTAT,'Y');
```

Chapter 14: Creating Charts

This chapter explains how to create and define charts, which provide a graphic display of scheduled tasks, using the **GANTT** program. These charts consist of four principal elements:

- Time range
- Employees/resources
- Tasks*
- Task dependencies (applicable only in an industrial setting)

The **GANTT** program supports three kinds of charts:

- **Gantt:** The X-axis is a variable timeline; the Y-axis consists of employees/resources.
- **Calendar:** The X-axis is a 24-hour timeline; the Y-axis consists of a main scale displaying days and a sub-scale displaying employees/resources.
- **Group Schedule:** The X-axis is a 24-hour timeline; the Y-axis consists of a main scale displaying employees/resources and a sub-scale displaying days.

The user can perform the following operations in a chart:

- Add new tasks
- Update/delete existing tasks
- View task details
- View employee details
- Toggle between chart types (Gantt/Calendar/Group Schedule)
- Vary the timeline
- Access the task in ***Priority***
- Access the employee record in ***Priority***

Defining a New Chart

To create a new chart, you need to define a procedure that activates the **GANTT** program. You must also define an interface to the appropriate form, to be driven by the **GANTT** program when updating or adding fields.

A procedure for building charts consists of two distinct sections:

Section 1: Begins with INPUT and/or SQLI steps and ends with the activation of the **GANTT** program.

Section 2: Query steps for data retrieval.

These two sections are separated by an END step.

* The term “task” is used throughout this chapter to represent any kind of operation (e.g., work order) that can be displayed in a chart.

Section 1: User Input and Activation of the GANTT Program

As in other procedures, this section requires user input, to determine which resources to display and during what time period. When run by Direct Activation, there is no need for an INPUT step, but you do need to initialize the necessary parameters (e.g., display mode, time range) in an SQLI step.

Whether you run the **GANTT** procedure by Direct Activation or from the menu, you must define a parameter of **FILE** type, indicating a linked file of records. This parameter is used to execute a link to the table on which the procedure will run. All queries executed by the **GANTT** program, in which data is retrieved from that table, are executed against the linked table (this is similar to passing a linked table to a report).

When the procedure is run by Direct Activation, you can define a specific task as "selected", by passing the task identifier in question to the **GANTT** program. In such a case, the chart opens with the specified task displayed. It is recommended to define a distinct and vivid display color for this task.

The following parameters must be defined for the **GANTT** program, in the following order:

1. The current procedure name (in order to retrieve queries from the second section).
2. *From Date*: The date from which chart data should be displayed.
3. *To Date*: The date until which chart data should be displayed
4. **LINK** file: Usually the employee/resource table from the procedure's INPUT step. You can also execute calculations before running the **GANTT** program, and link the table containing the calculation results. **Note:** If you indicate the **USERGANTT** table, the procedure does not execute a **LINK** to the table, but rather uses the original table.
5. The name of the linked table.
6. The name of the interface that updates/adds tasks.
7. The name of the form which the user can access for task details.
8. The name of the form which the user can access for employee details.
9. The flag permitting/preventing revisions to the chart (0/1).
10. The flag permitting/preventing additions to the chart (0/1).
11. The default display option (1 = Gantt, 2 = Calendar, 3 = Group Schedule,
    0 = the last display viewed by the current user, 4 = either the Calendar or Group Schedule display, whichever was last viewed by the current user).
12. Identifier of the selected task.
13. Identifier of the employee/resource assigned to the selected task.
14. – 17. Record null parameters for these positions.
18. An additional identifier, whose value appears in the variable **OTHERID**.
19. A second additional identifier, whose value appears in the variable **OTHERID2**.
20. The chart title.
21. The flag determining whether the chart is multi-company (0 = No, 1 = Yes).

Section 2: Defining Parameters

The following is the list of parameters to be defined in the second section of the procedure (all steps are *Type C*; their order is not important). A detailed explanation of each step is provided below the table.

Name of the Step	Returns
RESOURCE	The list of employees to display
RESOURCE_DETAILS	The details of a specific employee
TASKS	The list of tasks to display
TASK_DETAILS	The details of a specific task
TASK_TEXT	The text of a specific text
TASK_INSERT	Opening/updating a task via a form
TASK_EDIT	Query to define input fields in the dialogue box
TASK_REFRESH	Updated display of task details
WORKHOURS	Office hours for each day of the week
DAYSOFF	Non-working days
RESOURCE_WORKHOURS	Work hours per employee (instead of the previous two steps)
RELATIONS	Task dependencies (applicable only in an industrial setting)
RESOURCE_CHOOSE	Employee Choose list
RESOURCE_CHOOSE2	Additional Choose list (2)
RESOURCE_CHOOSE3	Additional Choose list (3)
RESOURCE_UPDATE	Update after choosing an employee
RESOURCE_UPDATE2	Update after additional Choose list (2)
RESOURCE_UPDATE3	Update after additional Choose list (3)
TASK_PRINT	Preparation of a **LINK** file before producing reports

Defining Employees (RESOURCE)

Fields to retrieve: employee identifier, employee name, target value, a sorting column

Query variables: FROMDATE, TODATE.

Notes:

Always retrieve values for all four fields.

You may choose not to use the variables to define your query conditions.

You may choose not to use the sorting column (in which case you retrieve a null value); its purpose is to allow you to sort records by something other than the employee name.

Example:

```
SELECT USER,USERNAME,USERLOGIN,1 FROM USERS
WHERE USER>0
ORDER BY USERLOGIN;
```

Retrieving Details of a Specific Employee (RESOURCE_DETAILS)

Fields to retrieve: Any desired employee details from the **USERS** or **USERSB** form.

Query variables (required): RESOURCEID.

Note: You can dynamically define procedure messages to serve as field titles for the returned values by using the # symbol for each field, followed by a message number.

Example:

```
SELECT USERLOGIN,SNAME,EMAIL,ADDRESS,SQL.DATES AS '#20'
FROM USERS,USERSB
WHERE USERS.USER = :RESOURCEID
AND USERS.USER = USERSB.USER;
```

Defining Tasks (TASKS)

Fields to retrieve: task identifier, employee identifier, task description, from date/time, to date/time, target value, display color

Query variables

- Required: FROMDATE, TODATE
- Optional: SELECTEDID

Notes:

Retrieve values for all of the above fields.

The date field must also include the time (DATE 14).

The display color is a value from the **COLORS** table (use 0 for no color).

Example:

```
SELECT DIARY,USER,TEXT
(CUSTNOTE <> 0 ? ITOA(CUSTNOTE,0,USECOMMA): ''),
CURDATE+STIME,CURDATE+ETIME,(DIARY = :SELECTEDID ? 1 : 0)
FROM DIARIES
WHERE CURDATE BETWEEN :FROMDATE AND :TODATE
AND USER = (SELECT USER FROM USERS !);
```

Retrieving Task Details (TASK_DETAILS)

Fields to retrieve: Any desired task or employee details from the **DIARIES** or **USERS** form.

Query variables (required): TASKID

Note: You can dynamically define procedure messages to serve as field titles for the returned values by using the # symbol for each field, followed by a message number.

Example:
```
SELECT DIARIES.TEXT AS '#19',USERS.USERNAME,
DIARIES.CURDATE,DIARIES.STIME,DIARIES.ETIME
FROM DIARIES,USERS
WHERE DIARIES.DIARY = :TASKID
AND DIARIES.USER = USERS.USER;
```

Retrieving Task Text (TASK_TEXT)

Fields to retrieve: Text, line order (**ORD**)

Query variables (required): TASKID

Example:
```
SELECT TEXT, TEXTORD FROM CUSTNOTESTEXT
WHERE CUSTNOTE = :TASKID
AND TEXTLINE > 0
ORDER BY TEXTORD;
```

Adding Tasks to the Chart (TASK_INSERT)

Record a trigger that inserts values in the fields of the designated form when a new task is opened.

Query variables (required): RESOURCEID, TASKDATE, TASKETIME, TASKSTIME, TASK.

Notes:

If this step is included, it will be possible to add and update tasks in the load form designated for the interface used to update/add tasks (defined in parameter 6). If it is not, it will still be possible to add and update tasks using the dialogue box.

In order to add or update tasks using either method, parameters must have been defined to permit additions or revisions to the chart (parameters 9 and 10, respectively).

Example:
```
:$.CURDATE = 0+ :TASKDATE;
:$.STIME = 0+ :TASKSTIME;
:$.ETIME = 0+ :TASKETIME;
SELECT USERLOGIN INTO :$.USERLOGIN
FROM USERS
WHERE USER = 0+ :RESOURCEID;
```

Defining Input Fields for the Dialogue Box (TASK_EDIT)

Fields to retrieve: Whatever is needed.

Query variables (required): RESOURCEID, TASKID, TASKSDATE, TASKEDATE, TASKSTIME, TASKETIME.

Notes:

The fields defined in this query automatically become input fields in the Add/Update Tasks dialogue box.

Before recording the query you must list in a note the fields in the **GENERALLOAD** table to which the input values will be transferred. Obviously, this list must correspond to the fields retrieved in the query.

You must make sure that the query does not fail (even when opening a new task, when the identifier is 0).

You can define mandatory fields by adding the letter M in parentheses after the name of the field in the relevant note.

Example:

```
/* Load fields: TEXT,DATE1(M),INT3,DATE2,INT4,
TEXT5,TEXT6,TEXT4,TEXT1,TEXT2,TEXT3,TEXT8 */
SELECT CUSTNOTESA.SUBJECT,
(:TASKID <> 0 ? CUSTNOTES.CURDATE : :TASKSDATE) AS '#20,'
(:TASKID <> 0 ? CUSTNOTES.STIME : :TASKSTIME) AS '#21,'
(:TASKID <> 0 ? CUSTNOTESA.TILLDATE : :TASKEDATE) AS '#27,'
(:TASKID <> 0 ? CUSTNOTES.ETIME : :TASKETIME) AS '#22,'
USERS.USERLOGIN AS '#24',USERS2.USERLOGIN AS '#26,'
CUSTOMERS.CUSTNAME,PHONEBOOK.NAME AS #25', CUSTNOTESA.LOCATION
FROM CUSTNOTES,CUSTNOTESA, CUSTOMERS, CUSTNOTETYPES,
PHONEBOOK,USERS,USERS USERS2
WHERE CUSTNOTES.CUSTNOTE = :TASKID
AND CUSTNOTES.CUSTNOTE = CUSTNOTESA.CUSTNOTE
AND CUSTNOTES.CUST = CUSTOMERS.CUST
AND CUSTNOTES.PHONE = PHONEBOOK.PHONE
AND CUSTNOTES.CUSTNOTETYPE = CUSTNOTETYPES.CUSTNOTETYPE
AND USERS.USER =
(:TASKID = 0 ? 0+ :RESOURCEID : CUSTNOTES.CUSER)
AND CUSTNOTESA.CUSER2 = USERS2.USER;
```

Updating the Display (TASK_REFRESH)

Fields to retrieve: task description, target value, from date/hour, to date/hour, color.

Query variables (required): RESOURCEID, TASKID

Example:

```
SELECT D.TEXT,ITOA(CUSTNOTES.CUSTNOTE,0,USECOMMA),
D.CURDATE+D.STIME,D.CURDATE+D.ETIME,0
FROM DIARIES D,CUSTNOTES
WHERE CN.CUSTNOTE = :TASKID
AND D.CUSTNOTE = CN.CUSTNOTE
AND D.USER IN (-9999,:RESOURCEID);
```

Office Hours (WORKHOURS)

Fields to retrieve: day, from hour, to hour

Query variables (required): RESOURCEID

Example:
```
SELECT WDAY,STARTT,ENDT FROM USERTMTBL
WHERE USER = :RESOURCEID;
```

Non-working Days (DAYSOFF)

Fields to retrieve: from hour

Query variables (required): CURDATE

Notes:

The query acts separately on each day within the designated date range.

On a regular workday the query should fail.

On holidays the query should return 0.

On days preceding holidays the query should return the time work ends.

Example:
```
SELECT FROMTIME FROM OFFICECLOSED
WHERE CURDATE = :CURDATE;
```

Employee Work Hours (RESOURCE_WORKHOURS)

Fields to retrieve: from date/hour, to date/hour

Query variables (required): RESOURCEID, FROMDATE, TODATE

Notes:

The query acts separately on each employee.

The query returns all work hours performed by the employee in the desired date range (i.e., more than one record).

The date field must also include the time (DATE 14).

This step can be used in place of the previous two steps (WORKHOURS and DAYSOFF).

Example:
```
SELECT WDATE+FROMTIME,WDATE+TOTIME FROM WORKHOURS
WHERE USER = :RESOURCEID
AND WDATE BETWEEN :FROMDATE AND :TODATE;
```

Task Dependencies (RELATIONS)

Task dependency is defined as a relationship in which the start or finish date of a task depends on another task.

Fields to retrieve: predecessor task identifier, successor task identifier, display color

Query variables (required): FROMDATE, TODATE

Note: This step is only applicable in an industrial setting.

Choose List for Employees (RESOURCE_CHOOSE)

Example:

```
SELECT USERNAME,USERLOGIN
FROM USERS,USERSB
WHERE USERS.USER = USERSB.USER
AND USERSB.SERVFLAG = 'Y'
AND USERSB.INACTIVE <> 'Y'
ORDER BY 1;
```

Additional Choose Lists (RESOURCE_CHOOSE2, RESOURCE_CHOOSE3)

Example:

```
SELECT GROUPDES,GROUPNAME
FROM UGROUPS
WHERE UGROUP <> 0
AND INACTIVE <> 'Y'
ORDER BY 1;
```

Update After Choosing an Employee (RESOURCE_UPDATE)

Query variables: CHOOSEVALUE, GANTTEXEC

Example:

```
INSERT INTO USERGANTT(EXEC,USER,RESOURCEID)
SELECT 0+:GANTTEXEC,SQL.USER,USER FROM USERS
WHERE USERLOGIN = :CHOOSEVALUE;
```

Update After Additional Choose Lists (RESOURCE_UPDATE2, RESOURCE_UPDATE3)

Query variables: CHOOSEVALUE, GANTTEXEC

Example:

```
INSERT INTO USERGANTT(EXEC,USER,RESOURCEID)
SELECT 0+:GANTTEXEC,SQL.USER,USER FROM USERGROUP
WHERE UGROUP = (SELECT UGROUP FROM UGROUPS
WHERE GROUPNAME = :CHOOSEVALUE);
```

Preparing the LINK File Before Producing Reports (TASK_PRINT)

Query variables: ZOOMVALUE, TASKID

Note: In order to print reports and/or documents from within the chart, add the print procedures as additional steps in the chart procedure.

Example:
INSERT INTO CUSTNOTES SELECT * FROM CUSTNOTES ORIG
WHERE CUSTNOTE = :TASKID;

Procedure Messages

As mentioned previously, you can dynamically define procedure messages to serve as field titles for retrieved task or employee details. Such messages should be assigned numbers greater than 20.

Messages 1 through 20 are already used by the **GANTT** program for various display titles. Consequently, when defining a new procedure you should always start by filling in the first twenty messages. The following table explains how these messages are used:

No.	Explanation	Example
1	Chart name	Calendar
2	Chart title	Calendar
3	Title while initializing employees	Loading employees…
4	Title while initializing tasks	Loading tasks…
5	Title while initializing dependencies	Loading dependencies…
6	Title for adding a task	New Appointment
7	Title for updating a task	Update Appointment
8	Title for the subject field when updating/adding a task	Subject
9	Title for employees/resources	Employees
10	Error message when adding a new task	Do not add a task
11	Error message when updating a task	Do not update a task
12	Error message when deleting a task	Do not delete a task
13	Title for the *Resource Search* dialogue box	Search for Employee/Date
14	Title for the *Resource Search* field	Employee
15	Title for the employee Choose list	Employees
16	Title for additional Choose list (2)	Groups*
17	Title for additional Choose list (3)	Team leaders*
18	Title for activating the employee Choose list	Choose an employee
19	Title for activating additional Choose list (2)	Choose a group*
20	Title for activating additional Choose list (3)	Choose a team leader*

* Different lists may be defined, at the discretion of the programmer. For example, in charts used to schedule technicians, you can allow users to retrieve records by service call type, rather than by team leader name.

Defining the Interface for Updating/Adding Tasks

The interface is defined against the **GENERALLOAD** table. Define an interface for any form against this table, and record the name of the interface as an argument in the call to the **GANTT** program. The interface will always run when

the table contains a single record whose code is **1** (in the *Form Load Designer*).

The values for adding/updating a task appear in the table in the following fields:

Field	Value
INT1	Task identifier
INT2	Employee/resource identifier
INT3	From hour
INT4	To hour
INT5	Previous employee/resource identifier (before update)
DATE1	From date (DATE 8)
DATE2	To date (DATE 8)
DATE3	From date/hour (DATE 14)
DATE4	To date/hour (DATE 14)

Notes:

You can use either separate fields for date and time, or a single field for both.

When the user adds a new task to the chart, the interface runs with task identifier **0**.

In addition to the constant fields, which appear in the above chart, values are transferred to additional fields, as defined in the TASK_EDIT step.

Chapter 15: Advanced Programming Tools

The purpose of this chapter is to demonstrate usage of advanced programming tools available in ***Priority***. These tools can be helpful in meeting specific programming needs. Before you begin, make sure that you are familiar with the basics (e.g., how to create a procedure or a form interface), as described in earlier chapters.

Working with the Priority Web Interface

Advanced programming for the ***Priority*** web interface is basically the same as programming for the Windows interface. However, when working with the web interface you must keep in mind the different method used to access the ***Priority*** server. As a result, the following exceptions apply:

- You cannot use the EXECUTE command to run entities that require input from the user or that display output. This is because the entity is executed on the server and any interface opened will be displayed on the server rather than on the local computer.
- You cannot add the following syntax to a procedure step in order to open a form when running a procedure:
  EXECUTE WINFORM 'ORDERS';
  Instead, add a new procedure step with the desired *Entity Name* and the *Type* **F**.
- You cannot add the following syntax to a procedure step in order to execute a document when running a procedure:
  EXECUTE WINACTIV '-P', 'WWWSHOWORDER', 'ORDERS', :TMPORDERS;
  LABEL 199;
- In procedures that load or export data, add the UPLOAD or DOWNLOAD step to upload/download the file in question from the local computer to the server (or vice versa). Specify the file name in the UPLOAD/DOWNLOAD procedure step.
- If the value in the *Application* column is 4 characters long and ends in 0 (zero), the entity in question will not be displayed when working with the web interface.

Example: See the UPLOAD procedure step in the **LOADFNC1** procedure and the DOWNLOAD step in the **ULOADFNC** procedure.

Note: This step is not necessary when opening or saving a file using an INPUT parameter of type **CHAR** that is flagged to display a browse button (in the *Procedure Parameter Extension* sub-level form).

Running a Procedure/Report from an SQLI Step or Form Trigger

You can execute a procedure from an SQLI step of another procedure by executing any of the following commands: **WINACTIV**, **ACTIVATE** or **ACTIVATF**.

This is useful, for example, when you want to run a report and send it to recipients via e-mail (see below).

The difference between the **WINACTIV** command and the **ACTIVATE** and **ACTIVATF** commands is that **WINACTIV** has a user interface, meaning that you can define a progress bar and/or messages that require a response from the user (using a **PRINTF**, **PRINTCONTF** or **CHOOSEF** command) and these will be visible to users while the procedure is running, whereas the **ACTIVATE** and **ACTIVATF** commands will not display these elements. As such, the **WINACTIV** command cannot be used when working with the ***Priority*** web interface.

The difference between the **ACTIVATE** and **ACTIVATF** commands is that **ACTIVATE** runs an *.exe* file whereas **ACTIVATF** runs a *.dll* file. In other words, a new process is created when the **ACTIVATE** command is used, whereas a procedure that is activated by the **ACTIVATF** command is executed in the same process as the form or procedure from which it is run.

Reports can be executed using the **WINACTIV** command only.

When using the **WINACTIV**, **ACTIVATE** or **ACTIVATF** commands, you can add two parameters for a linked table: the Table Name and the linked file (see examples below).

Example – Executing a Report: To execute the **OPENORDIBYDOER** report for a specific customer, the following code would be used:

```
:F = ../..output.txt';
SELECT SQL.TMPFILE INTO :CST FROM DUMMY;
LINK CUSTOMERS TO :CST;
GOTO 299 WHERE :RETVAL <= 0;
INSERT INTO CUSTOMERS SELECT * FROM CUSTOMERS O WHERE CUSTNAME =
'250';
UNLINK CUSTOMERS;
EXECUTE WINACTIV '-R', 'OPENORDIBYDOER','CUSTOMERS', :CST;
LABEL 299;
```

Example – Executing a Procedure: You have defined a special status for price quotes; whenever a quote receives this status, you want to open a sales order automatically based on that quote. To do so, create a custom POST-UPDATE trigger that checks whether the new status assigned to the quote is the special status and, if so, executes the *Open Order* procedure (**OPENORDBYCPROF**) using any of the three commands mentioned earlier. In the current example, the **ACTIVATF** command is used:

```
GOTO 10099 WHERE :$. CPROFSTAT <> :SPECIALSTATUS;
SELECT SQL.TMPFILE INTO :FILE FROM DUMMY;
LINK CPROF TO :PAR;
GOTO 10099 WHERE :RETVAL <= 0;
INSERT INTO CPROF SELECT * FROM CPROF O WHERE PROF = :$.PROF;
UNLINK CPROF;
EXECUTE ACTIVATF '-P', 'OPENORDBYCPROF', 'CPROF', :FILE;
LABEL 10099;
```

Running a Report and Sending it by E-mail

You might want to create a program that runs a report and sends it to recipients via e-mail. This is useful, for instance, when you write a procedure that runs a form interface, you execute this procedure via the Tabula Task Scheduler, and you want to send one of the users the errors report created by the form interface.

The following code runs a report and then sends the results in an e-mail attachment.

```
SELECT SQL.TMPFILE INTO :TMP FROM DUMMY;
LINK ERRMSGS TO :TMP;
GOTO 99 WHERE :RETVAL <= 0;
INSERT INTO ERRMSGS SELECT *
FROM ERRMSGS O WHERE USER = SQL.USER
AND TYPE = 'i';
GOTO 90 WHERE :RETVAL <= 0;
/* to send the report as an attachment to a Priority mail recipient */
:MAILER = SQL.USER;
EXECUTE WINACTIV '-R', 'INTERFACEERR', 'ERRMSGS', :TMP,'-u',
:MAILER;
/* to send the report as an attachment to a Priority group, defined in
the UGROUPS form */
:GROUPNAME = 'mailGroup';
EXECUTE WINACTIV '-R', 'INTERFACEERR', 'ERRMSGS', :TMP,'-g',
:GROUPNAME;
/* to send the report as an attachment to an external recipient */
:EMAIL = 'example@example.com';
EXECUTE WINACTIV '-R', 'INTERFACEERR', 'ERRMSGS', :TMP,'-e',
:EMAIL;
LABEL 90;
UNLINK ERRMSGS;
LABEL 99;
```

Alternatively, you can redirect the report results to a tab-delimited text file. In this case, you can use **ACTIVATF**. For example, the following code saves the output of the **OPENORDIBYDOER** report as a text file (tab-delimited) and then sends the results as an e-mail attachment.

```
:F = ../..output.txt';
SELECT SQL.TMPFILE INTO :CST FROM DUMMY;
LINK CUSTOMERS TO :CST;
GOTO 299 WHERE :RETVAL <= 0;
INSERT INTO CUSTOMERS SELECT * FROM CUSTOMERS O
WHERE CUSTNAME = '250';
UNLINK CUSTOMERS;
EXECUTE ACTIVATF '-x',:F, '-R',
'OPENORDIBYDOER','CUSTOMERS', :CST;
LABEL 299;
MAILMSG 5 TO EMAIL 'demo@demo.com' DATA, :F;
```

You can also redirect the report results to an MS-Excel file. This command takes two parameters – the Excel file without a suffix and the **TEMPLATE** number from the **EXCELTEMPLATES** table. For example, the following code saves the **OPENORDIBYDOER** report as an Excel file.

```
EXECUTE ACTIVATF '-P', 'ORGUNITS', '-X', 'c:\temp\cur', 444;
```

Program to Open a Form and Retrieve a Record

You can use SQL commands to open a given form and retrieve a given record. This is useful, for instance, when you want to create a new Direct Activation from the *Sales Orders* form that will open a customer shipment. In this example, once the document is successfully opened, the *Customer Shipments* form opens and the new document is retrieved automatically.

The following gives some basic guidelines for creating such a Direct Activation:

1. Define an interface (e.g., **YUVV_OPENDOC_D**) using the **GENERALLOAD** load table to open the **DOCUMENTS_D** form. In the interface definition, the **TEXT1** column will update the **ORDNAME** column in the **DOCUMENTS_D** form.
2. Create a procedure that runs the interface, and make it a Direct Activation from the **ORDERS** form. The procedure should include an INPUT step with a PAR parameter of **LINE** type, taken from the **ORDERS** table (where *Column Name* = **ORDNAME**). In the procedure, record the following in the SQLI step:

```
LINK ORDERS TO :$.PAR;
ERRMSG 1 WHERE :RETVAL <= 0;
:ORDNAME = '';
SELECT ORDNAME INTO :ORDNAME FROM
ORDERS WHERE ORD <> 0;
UNLINK ORDERS;
ERRMSG 2 WHERE :ORDNAME = '';
LINK GENERALLOAD TO :$.GEN;
ERRMSG 1 WHERE :RETVAL <= 0;
INSERT INTO GENERALLOAD(LINE,RECORDTYPE,TEXT1)
VALUES(1,'1',:ORDNAME);
EXECUTE INTERFACE 'YUVV_OPENDOC_D',SQL.TMPFILE,
'-L',:$.GEN;
:DOCNO = '';
SELECT DOCNO INTO :DOCNO FROM DOCUMENTS
WHERE TYPE = 'D' AND DOC = (
SELECT ATOI(KEY1) FROM GENERALLOAD
WHERE LINE = 1 AND LOADED = 'Y');
UNLINK GENERALLOAD;
ERRMSG 3 WHERE :DOCNO = '';
:$.DNO = :DOCNO; /* :$.DNO will be used in the next step to open the
ORDERS form in a web client - see below */
GOTO 9 WHERE :SQL.NET = 1;
/* to open the form and retrieve newly created record in Windows
client: */
```

```
EXECUTE WINFORM 'DOCUMENTS_D','',:DOCNO, '','2';
LABEL 9;
```

If you are working with the ***Priority*** web interface, add another procedure step that is not executed in the Windows client (use the GOTO procedure step to skip it). In this additional step, the *Entity Name* = **ORDERS** and the *Type* = 'F'. In the *Procedure Parameters* sub-level form, add the parameter DNO (of **CHAR** type).

Creating a Printout of a Document

You can create a procedure that opens a document and embed within it a second procedure that prints out that document. For instance, you can build a procedure that opens sales order via a form interface and then displays the printout of the new sales orders.

The PRINTFORMAT Table

As the printout will be created automatically, there is no user input to determine the print format; rather you have to set it yourself. This is done by means of the **PRINTFORMAT** table, which always saves the last print format utilized by a given user for a given document. Thus, when the document runs, the system takes the print format to be displayed from this table. In order to ensure that your procedure always displays the desired print format, you have to update the relevant record in the **PRINTFORMAT** table prior to execution of the document.

Determining Available Print Formats

To find out which print formats are available for a given document, run the following SQL commands in **WINDBI**:

```
/* this code will show all print formats defined for the document */
SELECT * FROM EXTMSG WHERE EXEC = (
SELECT EXEC FROM EXEC WHERE TYPE = 'P' AND ENAME =
'WWWSHOWORDER') AND NUM < 0 FORMAT;
```

Setting the Print Format

At this point you should know the **EXEC** of the document you want to run and the number of the print format you want to display. In the SQLI step of the procedure that runs the form interface for the *Sales Orders* form, add the following commands after execution of the form interface:

```
/* this code defines the format that will be used to print the document;
in the current example, it is assumed that we want to use print format -
5 */
:EXEC = 0;
SELECT EXEC INTO :EXEC FROM EXEC WHERE TYPE = 'P' AND
ENAME = 'WWWSHOWORDER';
:PRINTFORMAT = -5;
UPDATE PRINTFORMAT SET VALUE = :PRINTFORMAT
WHERE EXEC = :EXEC AND USER = SQL.USER;
```

```
SELECT SQL.TMPFILE INTO :TMPORDERS FROM DUMMY;
LINK ORDERS TO :TMPORDERS;
GOTO 199 WHERE :RETVAL <= 0;
INSERT INTO ORDERS SELECT * FROM ORDERS O
WHERE ORD = (
SELECT ATOI(KEY1) FROM GENERALLOAD
WHERE RECORDTYPE = '1' AND LOADED = 'Y');
/* Remark: In this example I assume the form interface uses the
GENERALLOAD table and that RECORDTYPE = '1' is the RECORDTYPE
used for the ORDERS form.
Reminder: After a successful load, the AutoUnique value of each new
order is saved in the KEY1 column of the load table */
```

Executing the Document

At this point you can execute the document in one of three ways: you can display it on screen, you can send it directly to the default printer, or you can save it as a file (and send as a mail attachment).

To display the document on screen, continue the above code as follows:

```
EXECUTE WINACTIV '-P', 'WWWSHOWORDER', 'ORDERS',
:TMPORDERS;
LABEL 199;
```

To send the document to the default printer, continue the above code as follows:

```
EXECUTE WINACTIV '-P', 'WWWSHOWORDER', '-q', 'ORDERS',
:TMPORDERS;
LABEL 199;
```

When sending the document to the default printer, if you want to print more than 1 copy, use the flag '–Q' instead of '–q'. For example, to print 2 copies, continue the above code as follows:

```
EXECUTE WINACTIV '-P', 'WWWSHOWORDER', '-Q',2, 'ORDERS',
:TMPORDERS; LABEL 199;
```

Note: The **WINACTIV** command can only be used when programming for the Windows interface; see p. 195 for details.

If you want to save the document as a file, you can choose between a few different file types.

- To save the document as an MHTML file (*..\..\orders.mht*), continue the above code as follows:

  ```
  :TMPOUT = '..\..\orders.mht';
  EXECUTE WINHTML '-d', 'WWWSHOWORDER', 'ORDERS',
  :TMPORDERS, '-m', :TMPOUT;
  ```

- To save the document as a standard HTML file (*..\..\orders.htm*), use the following code instead:

```
:TMPOUT = '..\..\orders.htm';
EXECUTE WINHTML '-d', 'WWWSHOWORDER', 'ORDERS',
:TMPORDERS, '-o', :TMPOUT;
```

- To save the document as a pdf file (*..\..\orders.pdf*), use the following code instead:

```
:TMPOUT = '..\..\orders.pdf';
EXECUTE WINHTML '-d', 'WWWSHOWORDER', 'ORDERS',
:TMPORDERS, '-pdf', :TMPOUT;
```

In a non-English system: If the document is in English (i.e., it is assigned the value *E* in the *HTML Document* column of the *Procedure Generator*), this should be indicated by adding the flag '-e' after the arguments of the linked file (i.e, after 'ORDERS', :TMPORDERS).

For example:

```
:TMPOUT = '..\..\orders_e.mht';
EXECUTE WINHTML '-d', 'WWWSHOWORDER_E', 'ORDERS',
:TMPORDERS, '-e', '-m', :TMPOUT;
```

Once you have generated the file, you can send an e-mail to a ***Priority*** user and attach the file:

```
:USER = SQL.USER;
MAILMSG 5 TO USER :USER DATA :TMPOUT;
```

Or you can send it to a ***Priority*** mail group:

```
:GROUPNAME = 'myGroup';
MAILMSG 5 TO GROUP :GROUPNAME DATA :TMPOUT;
```

or an external e-mail recipient:

```
:EMAIL = 'example@example.com';
MAILMSG 5 TO EMAIL :EMAIL DATA :TMPOUT;
LABEL 199;
```

Note: You have to record message number 5 for the procedure.

Executing a Document After Direct Activation

You might want to create a program that executes a document immediately after Direct Activation of a procedure from within a form (see Chapter 3, **Direct Activations**). If the desired procedure is run in the foreground (e.g., in order to display a message when it finishes running), the document should nevertheless be executed in the background so that users can continue working in the form during this time, as this process can take a few seconds. In such a case, the **WINHTML** program must be executed with the '-dQ' parameter.

For example, to print an invoice immediately after finalizing it via Direct Activation, include the following command in an SQLI step in the procedure (where the :IV variable represents the internal number of the invoice you wish to print):

```
EXECUTE BACKGROUND 'WINHTML', '-dQ','WWWSHOWAIV', :IV;
```

Setting a Number of Copies to Print

You also might want to create a program that generates multiple copies of a document by default, so that users need not update the number of copies each time the document is printed. The number of copies to be printed can be maintained in a custom form column.

For instance, say you want to create a procedure that prints multiple copies of a designated warehouse task, in which the number of copies is defined per warehouse. In order to achieve this, you can add a custom column to the **WAREHOUSES** form in which to define the desired number of copies. This value can then be retrieved when executing the document, in the HTMLCURSOR step of the procedure:

1. Add a custom column to the **WAREHOUSES** form (column name = **PRIV_NUMCOPIES**; column type = **INT**) that can receive any number between 1 and 7.
2. In the HTMLCURSOR step of the custom procedure, use the **DAYS** table to define a loop that generates the designated number of copies when executing the document:

```
SELECT WTASKS.WTASK, WTASKS.WTASKNUM,
(:$.SRT = 1 ? WTASKS.WTASKNUM :
(:$.SRT = 2 ? WAREHOUSES.WARHSNAME : ''))
FROM WTASKS, WAREHOUSES, DAYS
WHERE WTASKS.WTASK <> 0
AND WTASKS.WARHS = WAREHOUSES.WARHS
AND DAYS.DAYNUM BETWEEN 1 AND MAXOP(1,
WAREHOUSES.PRIV_NUMCOPIES)
ORDER BY 3, 2;
```

Notes:

If you wish to allow for a number of copies that is greater than 7, you can use a custom table rather than the **DAYS** table. However, any table you use must contain a fixed number of records, like the **DAYS** table.

If the user also specifies a number of copies when sending the document to the printer (e.g., 2), the total number of copies printed will be a product of both numbers (e.g., if **PRIV_NUMCOPIES** = 3, a total of 6 copies will be printed).

If this mechanism is used to generate multiple copies of a document in which only one original can be printed (such as an invoice or receipt), this mechanism does not override that restriction. In other words, additional copies that are generated in this fashion are still marked as copies.

Printing Files from a Procedure Step

To print files (e.g., attachments) from within a procedure, include the **PREXFILE** program in an SQLI step. The **PREXFILE** program receives several parameters:

- If you want to send the attachments directly to the default printer, add the flag '-d'. If you want to choose a printer, leave this parameter empty.
- The program can receive two optional parameters (**CHAR** type), which receive any description that should appear on a cover page, to be printed together with the attachment. If both parameters are empty, no cover page will be printed.
- The program receives a linked table (**STACK24** table, **FILE** type) which receives the file path(s) of the document(s) to be printed and the order in which multiple documents should be printed. This table contains the following columns:
  - **SORT_LINE —** Determines the printing order of the documents.
  - **TEXT1 —** The full path and name of the file to be printed.

Examples:
To create a new Direct Activation from the *Sales Orders* form, which prints any documents that are attached to the order, include an INPUT step with a PAR parameter of **LINE** type. In the procedure, record the following in the SQLI step:

```
SELECT SQL.TMPFILE INTO :STK FROM DUMMY;
LINK ORDERS TO :$.PAR;
ERRMSG 1 WHERE :RETVAL <= 0;
LINK STACK24 TO :STK;
ERRMSG 1 WHERE :RETVAL <= 0;
INSERT INTO STACK24(SORT_LINE,TEXT1)
SELECT EXTFILES.EXTFILENUM, EXTFILES.EXTFILENAME
FROM EXTFILES, ORDERS
WHERE EXTFILES.IV = ORDERS.ORD
AND EXTFILES.TYPE = 'O'
AND EXTFILES.EXTFILENUM > 0
AND ORDERS.ORD <> 0;
UNLINK STACK24;
UNLINK ORDERS;
```

To send the documents and cover page to the default printer, continue the above code as follows:

```
EXECUTE PREXFILE '-d', 'description 1', 'description 2', :STK;
```

To open a window to choose a printer from which to print the documents and cover page, use the following code instead:

```
EXECUTE PREXFILE 'description 1', 'description 2', :STK;
```

To send the documents to default printer without a cover page, use the following code instead:

```
EXECUTE PREXFILE '-d', '','', :STK;
```

Using a Form Interface to Duplicate Documents

Suppose you want to copy an entire sales order in order to create the same one for another customer. You can use the form interface to do this. In the following example, a new sales order is opened for a customer that is received as input, copying the order items, unit price, discount, ordered quantity, order item remarks and order remarks.

Begin by defining the following in the *Form Load Designer* and its sub-level forms (*System Management → Database Interface → Form Load (EDI)*):

Form Name	Title	Code (Record Type)
ORDERS	Sales Orders	1
ORDERSTEXT	Sales Orders - Remarks	2
ORDERITEMS	Order Items	3
ORDERITEMSTEXT	Order Items - Remarks	4

For the **ORDERS** form:

Load Table Column	Form Column Name	Order
TEXT1	CUSTNAME	1
TEXT2	DETAILS	2

For the **ORDERSTEXT** form:

Load Table Column	Form Column Name	Order
TEXT	TEXT	1

For the **ORDERITEMS** form:

Load Table Column	Form Column Name	Order
TEXT1	PARTNAME	1
REAL1	PRICE	2
REAL2	PERCENT	3
INT1	TQUANT	4

For the **ORDERITEMSTEXT** form:

Load Table Column	Form Column Name	Order
TEXT	TEXT	1

Next, create a new procedure with 2 input parameters. The first will be the customer for which you want to open the new order; the second will be the order you want to copy. The procedure will have 2 steps:

- An INPUT step with 2 input parameters:

Parameter Name	Pos	Width	Input (I/M)	Type	Column Name	Table Name
CST	0	0	I	LINE	CUSTNAME	CUSTOMERS
ORD	5	0	I	LINE	ORDNAME	ORDERS

- An SQLI step. Define a parameter called GEN (FILE type). The step query should look like this:

```
LINK CUSTOMERS TO :$.CST;
ERRMSG 1 WHERE :RETVAL <= 0;
:CUSTNAME = '';
SELECT CUSTNAME INTO :CUSTNAME FROM CUSTOMERS
WHERE CUST <> 0;
UNLINK CUSTOMERS;
ERRMSG 2 WHERE :CUSTNAME = '';
/* The following commands run the interface defined above, storing the
output in a linked file of the GENERALLOAD table, :$.GEN */
EXECUTE INTERFACE 'TEST_OPENSALESORD', SQL.TMPFILE, '-
o', '-L', :$.ORD, '-l', 'GENERALLOAD', :$.GEN;
LINK GENERALLOAD TO :$.GEN;
ERRMSG 1 WHERE :RETVAL <= 0;
UPDATE GENERALLOAD SET TEXT1 = :CUSTNAME
WHERE LINE = 1;
UNLINK GENERALLOAD;
EXECUTE INTERFACE 'TEST_OPENSALESORD',:$.MSG,
'-L',:$.GEN;
LINK GENERALLOAD TO :$.GEN;
ERRMSG 1 WHERE :RETVAL <= 0;
:ORD = 0;
SELECT ATOI(KEY1) INTO :ORD FROM GENERALLOAD
WHERE LINE = 1 AND LOADED = 'Y';
UNLINK GENERALLOAD;
ERRMSG 3 WHERE :ORD = 0;
:ORDNAME = '';
SELECT ORDNAME INTO :ORDNAME FROM ORDERS WHERE
ORD = :ORD;
/* The next command opens the Sales Orders form. */
EXECUTE BACKGROUND WINFORM 'ORDERS','',:ORDNAME, '','2';
```

Using the STACK_ERR Table to Store Interface Messages

When running the interface program using the '-m' option, each error message is broken up into several lines. Thus, for example, instead of the following message:

Line 1- Price Quotation A9000019: You cannot base the order on a price quotation of Draft status.

the user will receive 4 messages when the '-m' option is used:

1: Sales Rep Order Num
2: form 'Sales Orders'
3:Price Quotation A9000019
4: You cannot base the order on a price quotation of Draft status.

This can be problematic if the user tries to open two sales orders based on two different price quotations and both quotations have the Draft status. In

such a case, certain messages will not appear in the **ERRMSGS** table, since the unique key of the **ERRMSGS** table is: **USER**, **TYPE**, **MESSAGE**. In the above example, message number 4 will be the same in both cases and will therefore appear only once.

This problem is easily resolved by using the '-stackerr' option. This will insert all error messages into the **STACK_ERR** table, rather than using the **ERRMSGS** table to store interface messages. The original **LINE** value from the load table will be stored in the **INTDATA1** column of this table, and the error message will be stored in the **MESSAGE** column.

This option is also useful when you write a procedure that executes more than one interface. In such a case, the second interface would ordinarily delete any messages received by the first one. In order to maintain the messages generated by the first interface, use the '-stackerr' option and assign each interface a different parameter for the linked file. This will create a separate linked file of the **STACK_ERR** table for each interface you execute.

Example: To open sales orders based on price quotations, and then open a shipping document based on the new sales orders, the following code would be used. (In this example, it is assumed that the initial status for sales orders is flagged as *Allow Shipmt/ProjRep*.)

```
SELECT SQL.TMPFILE INTO :G1 FROM DUMMY;
SELECT SQL.TMPFILE INTO :G2 FROM DUMMY;
SELECT SQL.TMPFILE INTO :S1 FROM DUMMY;
SELECT SQL.TMPFILE INTO :S2 FROM DUMMY;
LINK GENERALLOAD ORD TO :G1;
GOTO 99 WHERE :RETVAL <= 0;
LINK GENERALLOAD DOC TO :G2;
GOTO 99 WHERE :RETVAL <= 0;
INSERT INTO GENERALLOAD ORD(LINE,RECORDTYPE,TEXT2)
SELECT SQL.LINE, '1', CPROFNUM FROM CPROF WHERE PDATE = SQL.DATE8;
EXECUTE INTERFACE 'OPENORDBYCPROF', SQL.TMPFILE, '-L',:G1, '-stackerr',:S1;
INSERT INTO GENERALLOAD DOC(LINE,RECORDTYPE,TEXT1)
SELECT SQL.LINE, '1', ORDNAME
FROM ORDERS, GENERALLOAD ORD
WHERE ORD.LOADED = 'Y'
AND ORDERS.ORD = ATOI(ORD.KEY1);
UNLINK GENERALLOAD ORD;
EXECUTE INTERFACE ' OPENDOC', SQL.TMPFILE, '-L',:G2, '-stackerr',:S2;
UNLINK GENERALLOAD DOC;
LINK STACK_ERR S1 TO :S1;
GOTO 99 WHERE :RETVAL <= 0;
SELECT * FROM STACK_ERR S1 FORMAT;
UNLINK STACK_ERR S1;
LINK STACK_ERR S2 TO :S2;
GOTO 99 WHERE :RETVAL <= 0;
SELECT * FROM STACK_ERR S2 FORMAT;
UNLINK STACK_ERR S2;
LABEL 99;
```

Executing SQL Queries Dynamically

SQL queries can be created and executed dynamically in forms or procedures using the command EXECUTE SQLI – which takes an ASCII file as a parameter.

Example: See step 20 of the **LOADMIGUSERS** procedure.

Executing SQL Server Code from Priority

SQL Server code can be executed from within ***Priority*** using the **SQLRUN** program. The program takes as a parameter an ASCII file containing the SQL Server code.

Example: See the **SQLCHECK** procedure.

Retrieving Data from the Client .ini File

When the user is working in the Windows interface of ***Priority***, you can retrieve information from the user's *.ini* file, by means of the **TABINI** program (which can be run from a form trigger or a procedure).

Note the following:

- The first parameter is the section in the *.ini* file from which you want to retrieve data. For example, to retrieve data from [Environment], pass 'Environment' as the first parameter.
- The second parameter is the line that you want to retrieve. For example, to get the path of the ***Priority*** directory, pass 'Priority Directory' as the second parameter.
- The third parameter is a LINK file of **GENERALLOAD** to which the program writes the retrieved data – in the first line (LINE = 1), in the **TEXT** column.
- If you pass empty strings as the first two parameters, the program will return the name of the *.ini* file itself.

Example:

```
SELECT SQL.TMPFILE INTO :A FROM DUMMY;
EXECUTE TABINI 'ENVIRONMENT', 'TABULA HOST', :A;
LINK GENERALLOAD TO :A;
SELECT TEXT FROM GENERALLOAD WHERE LINE = 1 FORMAT;
UNLINK GENERALLOAD;
```

Using Semaphores

If you want to prevent two users from running a given procedure or section of code concurrently, you can use a semaphore – a variable that can have one of two values (e.g., 0 or 1).

At the beginning of the code/procedure, check the value of the variable:

- If it is 0, this means no program is running this code. The variable should be updated to 1 and the code can be run. Upon completion, the variable should be set back to 0.
- If the variable value is 1, this means another program is running this section of code. You should either skip this section of code or have the user wait for the other program to finish.

While a customized table can be used for semaphores, the predefined **LASTS** table is suitable for this purpose.

LASTS has two columns:

- **NAME – CHAR** column (key)
- **VALUE – INT** column

In the **NAME** column, record a name for the semaphore, using the same prefix that you use for customizations.

Example: In the following code, the semaphore is a value in **LASTS** and the value in the **NAME** column is SDK_SEMAPHORE:

```
GOTO 1 FROM LASTS WHERE NAME = 'SDK_SEMAPHORE';
INSERT INTO LASTS(NAME) VALUES('SDK_SEMAPHORE');
LABEL 1;
UPDATE LASTS SET VALUE = 1 WHERE NAME = 'SDK_SEMAPHORE' AND VALUE = 0;
GOTO 99 WHERE :RETVAL <= 0;
...
...
...
UPDATE LASTS SET VALUE = 0 WHERE NAME = 'SDK_SEMAPHORE';
LABEL 99;
```

As there may be cases in which the program starts to run the section of code in question, but does not finish (e.g., power outage, system failure), you will need an additional procedure to "unlock" the program. This procedure should consists of a single step that resets the variable:

```
UPDATE LASTS SET VALUE = 0 WHERE NAME =
'SDK_SEMAPHORE';
```

Alternatively, you can run the code, even if it is "locked" by another user, once a certain span of time has elapsed since the procedure was "locked." This requires a customized table that contains the variable:

```
CREATE TABLE SDK_SEMAPHORES 'Semaphores' 0
NAME(CHAR, 48, 'Semaphore Name')
USER(INT, 8, 'User(id)')
UDATE(DATE, 14, 'Date')
UNIQUE(NAME);
```

Example: The following code allows the current user to run the section of code, even though the semaphore is set to 1, once 24 hours have elapsed since the code was last executed (under the assumption that, if locked that long, something must have prevented the program from reaching the section of code that resets the semaphore):

```
GOTO 1 FROM SDK_SEMAPHORES WHERE NAME = 'SDK_SEMAPHORE';
INSERT INTO SDK_SEMAPHORES(NAME) VALUES('SDK_SEMAPHORE');
LABEL 1;
UPDATE SDK_SEMAPHORES SET UDATE = SQL.DATE, USER = SQL.USER WHERE
NAME = 'SDK_SEMAPHORE' AND UDATE <= SQL.DATE - 24:00;
GOTO 99 WHERE :RETVAL <= 0;
...
...
...
UPDATE SDK_SEMAPHORES SET UDATE = 0, USER = 0 WHERE NAME =
'SDK_SEMAPHORE';
LABEL 99;
```

Activating Priority Entities from an External Application

To activate a ***Priority*** entity from a hyperlink that is embedded in an HTML file, the hyperlink should point to the following location (parameters are explained below):

priority:priform@FORMNAME:DOCUMENTNUM:COMPANY:TABINIFILE:LANG

To activate a ***Priority*** entity from the DOS command prompt (e.g., from within an external application), use the following syntax (parameters are explained below):

x:\priority\priform.exe priform@FORMNAME:DOCUMENTNUM:COMPANY:TABINIFILE:LANG

- If you are running this command from an external application, *x:\priority* is the folder in which ***Priority*** client files are located on the workstation.
- *FORMNAME* is the name of the ***Priority*** entity you want to activate (e.g., **ORDERS**). If this is not a form, it must be followed by the initial representing the entity type (M= menu; P = procedure; R = report). For example, to activate the *Project Reports* procedure, you would use the following syntax:
  priform@WWWDB_TRANSORDER_p.P
- *DOCUMENTNUM* is the ID number of the ***Priority*** record you want to retrieve in the designated form (not relevant when activating other types of entities). This is the value of the key column in the form's base table. For example, in the **ORDERS** form this would be the value of the **ORDNAME** column; in the **CUSTOMERS** form, it would be the value of the **CUSTNAME** column.
- *COMPANY* is the name of the ***Priority*** company in which you are executing the command.
- *TABINIFILE* is the name of the *tabula.ini* file (for example: *tabdev.ini*)
- *LANG* is the ID number of the language in which you want the form to open, as defined in the *Languages* form (e.g., for American English, specify 3).

Note: Unlike the Windows client, the web client does not provide a means of receiving a username and password as parameters when activating a form. If

users are not already logged into ***Priority*** on the workstation in question, they will need to enter a username and password before they can view any data.

Programs Interfacing Priority with External Systems

The following is a list of programs that may be run within ***Priority*** commands (e.g., form triggers, procedure steps), mainly for file management.

- Copy a file (**COPYFILE**):

  EXECUTE COPYFILE :f1,:f2;

- Download a file from the Internet (**COPYFILE**):

  EXECUTE COPYFILE '-i', :url, :tofile, [:msgfile];

- Move a file (**MOVEFILE**):

  EXECUTE MOVEFILE :f1,:f2;

- Delete a file (**DELWINDOW**):

  EXECUTE DELWINDOW 'f',:f1;

- Create a new folder (**MAKEDIR**):

  EXECUTE MAKEDIR :DIR;

- Display the date of a given file (**GETDATE**):

  EXECUTE GETDATE '*path/file_name*', :$.STK;

Example of use in a procedure step:

```
LINK STACK TO :$.STK;
ERRMSG 1 WHERE :RETVAL <= 0;
EXECUTE GETDATE 'path/file_name', :$.STK;
:FILEDATE = 0;
SELECT ELEMENT INTO :FILEDATE FROM STACK WHERE ELEMENT > 0;
UNLINK STACK;
```

- Display the size of a given file (**GETSIZE**):

  EXECUTE GETSIZE '*path/file_name*', :$.STK;

Example of use in a procedure step:

```
LINK STACK TO :$.STK;
ERRMSG 500 WHERE :RETVAL <= 0;
EXECUTE GETSIZE 'path/file_name', :$.STK;
:FILESIZE = 0;
SELECT ELEMENT INTO :FILESIZE FROM STACK WHERE ELEMENT > 0;
UNLINK STACK;
```

- Open a file using the default application for that file type (**SHELLEX**):

  ```
  :file = 'c:\test.doc';
  EXECUTE SHELLEX :file;
  /* if MS-Word is the default application for files of doc type, opens the
  c:\test.doc file in Word */
  ```

```
:file = 'www.google.com';
EXECUTE SHELLEX :file
 /* will open default browser and redirect to the URL specified in :file */
```

- Open a folder in Windows Explorer (**SHELLEX**):

```
:file = 'c:\temp';
EXECUTE SHELLEX :file;
```

- Return a random value in decimal or hexadecimal format (**PRANDOM**):

```
EXECUTE PRANDOM :file, :outtype;

/*to return a hexadecimal, use Y as the outtype parameter; to return a
decimal, specify anything other than Y */
```

Example of use in a procedure step:

```
SELECT SQL.TMPFILE INTO :STK1 FROM DUMMY;
SELECT SQL.TMPFILE INTO :STK2 FROM DUMMY;
EXECUTE PRANDOM :STK1, 'Y';
EXECUTE PRANDOM :STK2, '';
LINK STACK4 S1 TO :STK1;
GOTO 1 WHERE :RETVAL <= 0;
LINK STACK4 S2 TO :STK2;
GOTO 1 WHERE :RETVAL <= 0;
SELECT DETAILS AS 'RANDOM HEXA' FROM STACK4 S1 WHERE KEY = 1 FORMAT;
SELECT DETAILS AS 'RANDOM DECIMAL' FROM STACK4 S2 WHERE KEY = 1
FORMAT;
UNLINK STACK S1;
UNLINK STACK S2;
LABEL 1;
```

Output:

```
RANDOM HEXA
--------------------------------------------------------------------------------------------------------
81F5D3AB68A937242E9D888E84924A3C3F22B3261B119A69B49B4936

RANDOM DECIMAL
--------------------------------------------------------------------------------------------------------
189217014117615416177137247203401451172264225423531652 33
```

Filtering the Contents of a File

The **FILTER** program performs various manipulations on the contents of a specified text file. The examples below illustrate the different functions that the program can perform, such as finding and replacing all instances of a specific character, changing the character encoding (e.g., from ASCII to Unicode) and reversing the direction of text in the file. An explanation of the parameters that are used with this program appears beneath these examples.

- **FILTER** -replace *<Oldstring> <Newstring> {<Oldstring> <Newstring>} [Input Output]* — replaces the specified string or strings in the input file.
- **FILTER** [-r] *<Fromchar> <Tochar> <Targetchar>* [*Input Output*] [-M *Msgfile*] — converts each character in the input file that falls within a designated range of ASCII values (e.g., all uppercase letters from A to Z) to a new

character, using the following formula:
new character = original character + (Targetchar – Fromchar).

Example: The following command iterates through all characters in the input file that fall within the designated range (all uppercase letters from A to Z) and adds 32 to the ASCII value of these characters, thereby converting them to the corresponding lowercase letter:

```
EXECUTE FILTER 'A', 'Z', 'A', :INPUT, :OUTPUT;
```

You can also use this option to convert a tab-delimited file into a comma-delimited file (where '09' represents the ASCII value of the tab character):

```
EXECUTE FILTER '09', '09', ',', :INPUT, :OUTPUT;
```

- **FILTER** -unicode2ascii [*Input Output*] [-M *Msgfile*] — converts the input file from Unicode to ASCII.
- **FILTER** -ascii2unicode [*Input Output*] [-M *Msgfile*] — converts the input file from ASCII to Unicode.
- **FILTER** [-r] -pc [*Input Output*] [-M *Msgfile*] — converts the input file from DOS Hebrew to Windows Hebrew.
- **FILTER** -heb [*Input Output*] [-M *Msgfile*] — reverses the order of characters in Hebrew text.
- **FILTER** -addcr [*Input Output*] — When using an SQL query to export data from ***Priority*** tables, the '\n' (new line) character is added automatically to the end of each line. Use this option to add the '\r' (carriage return) character, as well, throughout the file (so that each line ends with the characters '\r\n').
- **FILTER** -unicode2utf8 [*Input Output*] — converts a UTF-16 file to UTF-8.
- **FILTER** -utf82unicode [*Input Output*] — converts a UTF-8 file to UTF-16.
- **FILTER** -trim [*Input Output*] — trims blank spaces at the beginning and end of each line in input file; also removes CR (carriage return) at the end of the line.

FILTER parameters

- '-r' — Use this option when you want the **FILTER** program to reverse the order of characters in the file (e.g., instead of 'abcd' you receive 'dcba').
- *Input*, *Output* — The input and output files to which the **FILTER** program refers.
- '-M', '*Msgfile*' — Use this option when you want the **FILTER** program to record error messages in the designated file.

Browsing the Contents of a Folder

The **FILELIST** program browses the contents of a specified folder. It is very useful, for example, when you want to create an automatic procedure (to be run by the Tabula Task Scheduler) that checks the contents of a folder and loads certain files from that folder into ***Priority*** tables.

The following code is used when you have an external program that creates files in one of your system folders, and the files contain data of new sales

orders that you need to load into ***Priority***. It is assumed that the files to be loaded are named as follows: *loadorder1201061355.load* (where the number represents the date and time the file was created).

```
:DIR = '../../tmpDir';
SELECT SQL.TMPFILE INTO :ST6 FROM DUMMY;
SELECT SQL.TMPFILE INTO :MSG FROM DUMMY;
EXECUTE FILELIST :DIR,:ST6,:MSG;
/* In the linked file of the STACK6 table, you will find all files and folders
under the input directory :DIR. */
LINK STACK6 TO :ST6;
GOTO 99 WHERE :RETVAL <= 0;
DECLARE NEWFILES CURSOR FOR
SELECT TOLOWER(NAME) FROM STACK6
WHERE TOLOWER(NAME) LIKE ' loadorder*';
OPEN NEWFILES;
GOTO 90 WHERE :RETVAL <= 0
:FILENAME = '';
:TOFILENAME = '../../system/load/Example.load';
LABEL 10;
FETCH NEWFILES INTO :FILENAME;
GOTO 85 WHERE :RETVAL <= 0;
:PATH = STRCAT(:DIR,'/',:FILENAME);
/* now the variable :path holds the filename */
/* there are 2 options to execute the DBLOAD */
/* option 1: */
EXECUTE COPYFILE :PATH, :TOFILENAME;
/* here you need to make sure you define the load Example.load */
EXECUTE DBLOAD '-L', 'Example.load';
/* option 2: add two more parameters to the DBLOAD program; these
parameters tell the DBLOAD program to load the file that comes after
the -i option */
EXECUTE DBLOAD '-L', 'Example.load', -i, :PATH;
LOOP 10;
LABEL 85;
CLOSE NEWFILES;
LABEL 90;
UNLINK STACK6;
LABEL 99;
```

Running an External Application (WINAPP)

The **WINAPP** command allows you to run any application (.exe file) and to define parameters for it (e.g., a ***Priority*** field). The command must be included in an SQLI step of a ***Priority*** procedure and should have the following structure:

- EXECUTE WINAPP
- The full path in which the external program is located
- The **-W** parameter (optional). This parameter causes **WINAPP** to wait until the external program is completed before returning to ***Priority***.

- The command that runs the application (i.e., the program name, with or without the .exe suffix)
- Any desired parameter for the external program
- Any ***Priority*** parameters that serve as input to the external program.

Strict attention must be paid to the proper punctuation in the command line (see examples below):

- Use a comma (,) to separate command segments.
- Record single quote marks (') before and after each command segment.
- Add a semi-colon (;) at the end of the command line.

Examples:

To run MS-Word:
EXECUTE WINAPP 'C:\Program Files\Microsoft Office\Office', 'WINWORD.EXE';

To open the *tabula.ini* file in Notepad and return to ***Priority*** only after Notepad is exited:
EXECUTE WINAPP 'C:\Windows', '-w','notepad','tabula.ini';

Executing *Priority* Commands from an External Application (WINRUN)

Using the **WINRUN** command, you can execute any ***Priority*** entity from the DOS command prompt. To do so, use the following syntax (parameters are explained below):

> *x*:\priority\bin.95\winrun ""*username password x*:\priority\system\prep *company* -nbg -err errfile *command arguments*

- *x* is the drive on which ***Priority*** is located. If you are not running this command from the server, *x* is the *network* drive on which ***Priority*** is located on the server.
- The second parameter is two double quote marks (").
- *company* is the name of the ***Priority*** company in which you are executing the command.
- **-nbg** — Use this option if you want the entity to run in the foreground, rather than the background.
- **-err** *errfile* — Use this option when executing a procedure using the **WINPROC** or **WINACTIV** commands if you want to have error messages sent to the specified error file instead of being displayed on screen.
- The command that runs the entity (e.g., **WINFORM**, **WINACTIV**), followed by the argument(s) required for the specified command. For example:
  - To open a form, use the **WINFORM** command, where the argument is the internal name of the form to be opened.
  - To run a procedure or report, use the **WINACTIV** or **WINPROC** commands, where the first argument is either **-P** for a procedure or **-R** for a report that is run without an accompanying procedure, and the second argument is the internal name of the procedure/report.

- To run an interface, use the **INTERFACE** command, where the first argument is the name of the interface and the second is the name of the temporary file in which to store load messages.

Notes:

The **WINRUN** command will run on the ***Priority*** installation referred to in the *C:\Windows\tabula.ini* file on the workstation from which the command is run. To work with a different *tabula.ini* file, add the command:
set TABULAINI=*xxx*.ini
where *xxx* is the name of the file in question.

When executing the **WINACTIV** command with the **-err** *errfile* option, only preliminary messages will be written to the specified file (i.e., messages that are not generated by the procedure itself, but from preliminary programs, such as those that check for problems with your license and/or user privileges). Messages generated by the procedure during runtime are written to the **ERRMSGS** table. Messages are recorded to this table for the **USER** that executed the **WINRUN** command and with the **TYPE** = 'V'.

Examples:

In the following examples, ***Priority*** is installed on the server in the **d** drive, which is mapped on the workstation as drive **p**. The **WINRUN** command is run in the **demo** company for the **tabula** user, whose password is **XYZabc1**.

To open the *Sales Orders* form from the workstation:
p:\priority\bin.95\winrun "" tabula XYZabc1 p:\priority\system\prep demo WINFORM ORDERS

To run the *Overnight Backflush-New Transact* program from the server:
d:\priority\bin.95\winrun "" tabula XYZabc1 d:\priority\system\prep demo WINACTIV -P BACKFLUSH_ONNEW

To run the interface that loads sales orders from the server:
d:\priority\bin.95\winrun "" tabula XYZabc1 d:\priority\system\prep demo INTERFACE LOADORDERS d:\priority\tmp\messages.txt

Parsing XML/JSON Files

In addition to reading data from an XML/JSON file via a form load (see Chapter 7), you can also use the **XMLPARSE** command.

When the file contains several instances per tab, include the *–all* parameter to parse the entire file. Omit it to limit results to the first instance of each tab.

Example:

```
SELECT SQL.TMPFILE INTO :OUTXMLTAB1 FROM DUMMY;
SELECT SQL.TMPFILE INTO :OUTXMLTAB2 FROM DUMMY;
SELECT SQL.TMPFILE INTO :MSG FROM DUMMY;
LINK INTERFXMLTAGS I1 TO :OUTXMLTAB1;
GOTO 500 WHERE :RETVAL <= 0;
LINK INTERFXMLTAGS I2 TO :OUTXMLTAB2;
GOTO 500 WHERE :RETVAL <= 0;
:FILE = '../../system/load/example.xml';
EXECUTE XMLPARSE :FILE, :OUTXMLTAB1, 0, :MSG;
EXECUTE XMLPARSE :FILE, :OUTXMLTAB2, 0, :MSG, '-all';
SELECT LINE, TAG, VALUE, ATTR FROM INTERFXMLTAGS I1 WHERE LINE > 0
FORMAT;
SELECT LINE, TAG, VALUE, ATTR FROM INTERFXMLTAGS I2 WHERE LINE > 0
FORMAT;
LINK INTERFXMLTAGS I1;
LINK INTERFXMLTAGS I2;
LABEL 500;
```

When the XML file is the one displayed below (see next page), results for the above two **EXECUTE** commands (without the *–all* parameter and with it) are as follows:

```
LINE     TAG                                                                                                VALUE
-------- -------------------------------------------------------------------------------------------------- --------------------
       1 DOCUMENTS/DOCUMENT/DOCNO                                                                           A001
       2 DOCUMENTS/DOCUMENT/CURDATE                                                                         20/09/16
       3 DOCUMENTS/DOCUMENT/DETAILS                                                                         First doc details
       4 DOCUMENTS/DOCUMENT/LINES/LINE/PARTNAME                                                             P1
       5 DOCUMENTS/DOCUMENT/LINES/LINE/TQUANT                                                               1
LINE     TAG                                                                                                VALUE
-------- -------------------------------------------------------------------------------------------------- --------------------
       1 DOCUMENTS/DOCUMENT/DOCNO                                                                           A001
       2 DOCUMENTS/DOCUMENT/CURDATE                                                                         20/09/16
       3 DOCUMENTS/DOCUMENT/DETAILS                                                                         First doc details
       4 DOCUMENTS/DOCUMENT/LINES/LINE/PARTNAME                                                             P1
       5 DOCUMENTS/DOCUMENT/LINES/LINE/TQUANT                                                               1
       6 DOCUMENTS/DOCUMENT/LINES/LINE/PARTNAME                                                             P2
       7 DOCUMENTS/DOCUMENT/LINES/LINE/TQUANT                                                               2
       8 DOCUMENTS/DOCUMENT/LINES/LINE/PARTNAME                                                             P3
       9 DOCUMENTS/DOCUMENT/LINES/LINE/TQUANT                                                               3
      10 DOCUMENTS/DOCUMENT/DOCNO                                                                           A002
      11 DOCUMENTS/DOCUMENT/CURDATE                                                                         20/09/16
      12 DOCUMENTS/DOCUMENT/DETAILS                                                                         Second doc details
      13 DOCUMENTS/DOCUMENT/LINES/LINE/PARTNAME                                                             P21
      14 DOCUMENTS/DOCUMENT/LINES/LINE/TQUANT                                                               1
      15 DOCUMENTS/DOCUMENT/LINES/LINE/PARTNAME                                                             P22
      16 DOCUMENTS/DOCUMENT/LINES/LINE/TQUANT                                                               2
      17 DOCUMENTS/DOCUMENT/LINES/LINE/PARTNAME                                                             P23
      18 DOCUMENTS/DOCUMENT/LINES/LINE/TQUANT                                                               3
```

```
<?xml version="1.0" encoding="UTF-16" standalone="no" ?>
- <DOCUMENTS>
  - <DOCUMENT>
      <DOCNO>A001</DOCNO>
      <CURDATE>20/09/16</CURDATE>
      <DETAILS>First doc details</DETAILS>
    - <LINES>
      - <LINE>
          <PARTNAME>P1</PARTNAME>
          <TQUANT>1</TQUANT>
        </LINE>
      - <LINE>
          <PARTNAME>P2</PARTNAME>
          <TQUANT>2</TQUANT>
        </LINE>
      - <LINE>
          <PARTNAME>P3</PARTNAME>
          <TQUANT>3</TQUANT>
        </LINE>
      </LINES>
    </DOCUMENT>
  - <DOCUMENT>
      <DOCNO>A002</DOCNO>
      <CURDATE>20/09/16</CURDATE>
      <DETAILS>Second doc details</DETAILS>
    - <LINES>
      - <LINE>
          <PARTNAME>P21</PARTNAME>
          <TQUANT>1</TQUANT>
        </LINE>
      - <LINE>
          <PARTNAME>P22</PARTNAME>
          <TQUANT>2</TQUANT>
        </LINE>
      - <LINE>
          <PARTNAME>P23</PARTNAME>
          <TQUANT>3</TQUANT>
        </LINE>
      </LINES>
    </DOCUMENT>
</DOCUMENTS>
```

Defining Word Templates for Specific Form Records

Suppose you have defined several Word templates in a given form, and you want to use a particular one when sending specific form records to Word. This can be achieved by adding a form column of **INT** type, whose name contains the string **AWORD** (e.g., **PRIV_AWORD**). This column receives the number of one of the Word templates defined for the form in question.

Note: Word templates are saved as form messages and assigned a negative number (hence, they do not appear in the *Error & Warning Messages* sub-level of the *Form Generator* form).

In the following example, several Word templates have been defined for the *Tasks* form, and you want to use the designated *Task Code* to determine which template is used for each task.

1. Add a new column to the **CUSTTOPICS** table: **PRIV_AWORD – INT**, 8, *Word Template*.
2. Add the following columns to the **CUSTTOPICS** form:

Form Column Name	Column Name	Table Name	Column ID	Join Column	Join Table	Join ID
PRIV_AWORD	PRIV_AWORD	CUSTTOPICS	0	NUM	TRIGMSG	5?
PRIV_MESSAGE	MESSAGE	TRIGMSG	5			0
PRIV_EXEC	EXEC	TRIGMSG	5			0

Note: The **TRIGMSG** table is outer-joined because a Word template can be deleted.

3. Hide the **PRIV_EXEC** column and mark the **PRIV_MESSAGE** column as read-only.
4. Create a PRE-FORM trigger (PRIV_PRE-FORM) in the **PRIV_EXEC** column, which retrieves the internal number of the **CUSTNOTESA** form:

```
:PRIV_EXEC = 0;
SELECT EXEC INTO :PRIV_EXEC FROM EXEC WHERE ENAME =
'CUSTNOTESA' AND TYPE = 'F';
```

5. Set the value of the **PRIV_EXEC** form column in the *Form Column Extension* sub-level form, in the *Expression/Condition* column:

```
= :PRIV_EXEC /* the variable was initiated in PRIV_PRE-FORM */
```

6. In the *Error & Warning Messages* form, add message 501:
"Select a value from the Choose list."
7. Add the following 3 triggers to the **PRIV_AWORD** form column:
   - PRIV_CHECK-FIELD

```
ERRMSG 501 WHERE :$.@ <> 0 AND NOT EXISTS(
SELECT * FROM TRIGMSG WHERE EXEC = :PRIV_EXEC AND
NUM = :$.@
AND NUM < 0);
```

   - PRIV_CHOOSE-FIELD

```
/* List of Word templates defined for Tasks (CUSTNOTESA) form */
SELECT MESSAGE, ITOA(NUM) FROM TRIGMSG WHERE EXEC =
:PRIV_EXEC AND NUM < 0 ORDER BY 2;
```

   - PRIV_POST-FIELD

```
:$.PRIV_EXEC = :PRIV_EXEC;
```

8. Add a hidden column to the **CUSTNOTESA** form:

Form Column Name	Column Name	Table Name
PRIV_AWORD	PRIV_AWORD	CUSTTOPICS

This column is filled in with the number of the template defined for the task code of the current task. Thereafter, when a user chooses to send a task to Word, if the task code of the task in question has been assigned a Word template, the information is sent directly to the designated template.

If the user flags the **All Displayed Records** option, those records with a different Word template than that defined for the current record are not sent to Word.

Custom Form Columns in the Business Rules Generator

The Business Rules Generator enables users to set up error, warning, e-mail or text messages (SMS) that are triggered automatically when certain conditions are met. When the action selected is *Send e-mail* or *Send text msg*, users can choose to send the e-mail/text message to any of the form columns appearing in the provided Choose list. If you want a custom form column to appear in this Choose list, it must meet ***one*** of the following conditions:

- The name of a calculated form column must contain the following string: **EMAIL** (e.g., **PRIV_ORDEMAIL**). The e-mail or text message will be sent to the address or phone number defined in that form column.
- A regular form column must be taken from one of the following tables/table columns:
  - **CUSTOMERS (CUSTNAME)**
  - **SUPPLIERS (SUPNAME)**
  - **AGENTS (AGENTNAME)**
  - **USERSB (SNAME)**
  - **PHONEBOOK (NAME)**
  - **USERS (USERLOGIN)**
  - **UGROUPS (GROUPNAME)**

  In such a case, the Business Rules Generator uses the e-mail address or phone number defined in the specified table: **CUSTOMERS.EMAIL/PHONE**, **SUPPLIERS.EMAIL/PHONE**, **AGENTS.EMAIL/PHONE**, **USERSB.EMAIL/CELLPHONE**, **PHONEBOOK.EMAIL/CELLPHONE**. For form columns taken from the **USERS** table, the e-mail/text message will be sent to the employee associated with the specified user (in the *Personnel File* form). For form columns taken from the **UGROUPS** table, the e-mail/text message will be sent to the members of the specified group.

Chapter 16: Built-in ODBC Functions for SQL Database

Introduction

Several ODBC functions have been written for ***Priority*** when using an MS-SQL database. This chapter contains a comprehensive list of string functions, number functions and date functions, together with examples of their usage.

String Functions

The following functions are performed on strings:

- **system.dbo.tabula_itoa** *(m,n,x,y)* — yields *m* as a string having *n* characters, where both values are integers (leading zeroes are added where necessary); *x* indicates whether to separate thousands (0 = no, 1 = yes) and *y* is the separator.

  ```
  SELECT system.dbo.tabula_itoa(50000,7,1,',') /* '0050,000' */
  SELECT system.dbo.tabula_itoa(50000,7,0,',') /* '0050000' */
  ```

- **system.dbo.tabula_rtoa** *(m,n,x,y)* — yields *m* (a real number) as a string, displaying *n* decimal places, where *x* is the thousands separator and *y* is the decimal point.

  ```
  SELECT system.dbo.tabula_rtoa(109999.35,2,',','.') /* '109,999.35' */
  SELECT system.dbo.tabula_rtoa(109999.35,2,'-',',') /* '109-999,35' */
  ```

- **system.dbo.tabula_atoi** (*string*) — outputs the designated string as an integer.

  ```
  SELECT system.dbo.tabula_atoi('100') /* 100 */
  ```

- **system.dbo.tabula_ator** (*string*) — outputs the designated string as a real number.

  ```
  SELECT system.dbo.tabula_ator('100.66') /* 100.66 */
  ```

- **system.dbo.tabula_stritoa** (*string, n*) — moves the decimal point *n* places to the left. When input is a string that contains one or more decimal points (which break up the string into parts), the function is applied separately to each part; this is useful for sorting columns that contain WBS codes.

  ```
  SELECT system.dbo.tabula_stritoa('1',3) /* 001 */

  SELECT system.dbo.tabula_stritoa('3.5.2',3) /* .003.005.002 */
  ```

- **system.dbo.tabula_strpiece** (*string, delimiter, m, n*) — for a given input string and delimiter (which breaks up the string into parts), retrieves *n* parts, beginning from the *m*th part.

  ```
  SELECT system.dbo.tabula_strpiece('hello-world','-',2,1) /* 'world' */

  SELECT system.dbo.tabula_strpiece('112.113.114-115','.',1,3) /*
  '112.113.114' */
  ```

Note: The following function is intended for users of ***Priority*** in Hebrew.

The tabula_hebconvert function performs automated conversion of Hebrew text formats in strings that contain both English and Hebrew letters.

This function does not appear in the default list of system functions for ***Priority*** on SQL Server. To add tabula_hebconvert to the list of system functions, select *Run Entity (Advanced)* from the *Tools* top menu in the Windows interface or the *Run* menu in the web interface, and run the **HEBCONV** program.

Example: The following command yields a string that contains both English and Hebrew letters. Note that the English letters and the numerals appear backwards.

```
SELECT demo.dbo.TRIALBAL.TRIALBALDES
FROM demo.dbo.TRIALBAL
WHERE demo.dbo.TRIALBAL.TRIALBALCODE = '888'
```

Result:)wen(154 סעיף

Using the tabula_hebconvert function, the output appears as follows:

```
SELECT system.dbo.tabula_hebconvert(demo.dbo.TRIALBAL.TRIALBALDES)
FROM demo.dbo.TRIALBAL
WHERE demo.dbo.TRIALBAL.TRIALBALCODE = '888'
```

Result: (new) 451 סעיף

Number Functions

The following functions are performed on numbers:

- **system.dbo.tabula_htoi**(*m*) — inputs a hexadecimal value and translates it to its corresponding integer.

  ```
  SELECT system.dbo.tabula_htoi('ff') /* 255 */
  ```

- **system.dbo.tabula_itoh**(*n*) — returns the hexadecimal value of *n*, where *n* is an integer.

  ```
  SELECT system.dbo.tabula_itoh(15) /* 'f' */
  ```

Shifted Integers

When working with shifted integers, there is often a need to convert them to real numbers. In ***Priority***, the REALQUANT function performs this conversion automatically.

There is currently no ODBC-compliant equivalent to the REALQUANT function. Therefore, when writing queries using SQL tools, it is necessary to execute the division manually. The number of places that the decimal point should be moved is determined by the value of the DECIMAL system constant (usually, 3).

Example:
```
SELECT QUANT, 0.0 + TQUANT / 1000,
FROM DEMO.DBO.ORDERITEMS WHERE ORD <> 0
/* assuming the DECIMAL constant = 3 */

/*In SSMS (MS SQL) using CAST()*/
SELECT QUANT, (CAST(TQUANT AS REAL) / 1000) AS TQUANT
FROM DEMO.DBO.ORDERITEMS WHERE ORD <> 0
```

Notes:

If the decimal precision is defined as 2 or 1, the integer will need to be divided by 100 or 10, accordingly.

To check the value of the DECIMAL system constant, execute the following query:
```
SELECT VALUE FROM DEMO.DBO.SYSCONST WHERE NAME = 'DECIMAL';
```

Date Functions

In ***Priority***, dates, times and days are stored in the database as integers, which correspond to the number of minutes elapsed since Jan. 1, 1988 (for example, Dec. 31, 1987 = -1440). These integers can be retrieved using the *SQL Development* (**WINDBI**) program by writing a query such as:

```
SELECT 0 + IVDATE FROM INVOICES FORMAT;
```

This enables you to perform calculations on dates.

Example: To calculate the day after an invoice date, use:
```
SELECT IVDATE + 24:00 FROM INVOICES FORMAT;
```

The following functions are performed on dates:

- **system.dbo.tabula_dateconvert** (*date*) — converts a ***Priority*** date to an SQL date.

  ```
  SELECT demo.dbo.INVOICES.IVDATE,
  system.dbo.tabula_dateconvert(demo.dbo.INVOICES.IVDATE)
  FROM demo.dbo.INVOICES;
  ```

- **system.dbo.tabula_bofyear** (*date*) — yields the date of the first day of the year.

  ```
  SELECT system.dbo.tabula_dateconvert(
  system.dbo.tabula_bofyear(demo.dbo.INVOICES.IVDATE))
  FROM demo.dbo.INVOICES;
  ```

- **system.dbo.tabula_eofyear** (*date*) — yields the date of the end of the year.

  ```
  SELECT system.dbo.tabula_dateconvert(
  system.dbo.tabula_eofyear(demo.dbo.INVOICES.IVDATE))
  FROM demo.dbo.INVOICES;
  ```

- **system.dbo.tabula_bofhalf** (*date*) yields the date of the first day of the six-month period (half a year) in which the date falls.

  ```
  SELECT system.dbo.tabula_dateconvert(
  system.dbo.tabula_bofhalf(demo.dbo.INVOICES.IVDATE))
  FROM demo.dbo.INVOICES;
  ```

- **system.dbo.tabula_eofhalf** (*date*) — yields the date of the end of the half-year.

  ```
  SELECT system.dbo.tabula_dateconvert(
  system.dbo.tabula_eofhalf(demo.dbo.INVOICES.IVDATE))
  FROM demo.dbo.INVOICES;
  ```

- **system.dbo.tabula_bofquarter** (*date*) — yields the date of the first day of the quarter.

  ```
  SELECT system.dbo.tabula_dateconvert(
  system.dbo.tabula_bofquarter(demo.dbo.INVOICES.IVDATE))
  FROM demo.dbo.INVOICES;
  ```

- **system.dbo.tabula_eofquarter** (*date*) — yields the date of the end of the quarter.

  ```
  SELECT system.dbo.tabula_dateconvert(
  system.dbo.tabula_eofquarter(demo.dbo.INVOICES.IVDATE))
  FROM demo.dbo.INVOICES;
  ```

- **system.dbo.tabula_bofmonth** (*date*) — yields the date of the first day of the month.

  ```
  SELECT system.dbo.tabula_dateconvert(
  system.dbo.tabula_bofmonth(demo.dbo.INVOICES.IVDATE))
  FROM demo.dbo.INVOICES;
  ```

- **system.dbo.tabula_eofmonth** (*date*) — yields the date of the end of the month.

  ```
  SELECT system.dbo.tabula_dateconvert(
  system.dbo.tabula_eofmonth(demo.dbo.INVOICES.IVDATE))
  FROM demo.dbo.INVOICES;
  ```

- **system.dbo.tabula_week** (*date, day year began*) — yields an integer comprised of the year (last digit of the year) and the number of the week in the year (two digits, between 01 and 53).

  ```
  SELECT system.dbo.tabula_week(demo.dbo.INVOICES.IVDATE, 3)
  FROM demo.dbo.INVOICES;
  ```

- **system.dbo.tabula_bofweek** (*week no., day year began*) — given a value for a week (the last two digits of a year and the number of a week in that year) and the day the year started, yields the date that week started.

  ```
  SELECT system.dbo.tabula_dateconvert(
  system.dbo.tabula_bofweek(0801, 1));
  ```

- **system.dbo.tabula_week6** (*date, day year began*) — yields an integer comprised of the year in 4 digits and the number of the week in the year (two digits, between 01 and 53).

```
SELECT system.dbo.tabula_week6(demo.dbo.INVOICES.IVDATE, 3)
FROM demo.dbo.INVOICES;
```

- **system.dbo.tabula_mweek** (*week, day year began*) — given a value for a week (the last two digits of a year and the number of a week in that year) and the day the year began, yields the number of the month in which that week falls.

```
SELECT system.dbo.tabula_mweek(0819,3);
```

- **system.dbo.tabula_dtoa** (*date, pattern, '<months>', '<days>'*) — converts a date to a string. If you are not displaying the names of the days or months, the last two parameters should be empty strings.

  See usage below.

- **system.dbo.tabula_atod** (*string, pattern*) — converts a string to a date.

  See usage below.

Date Pattern Components for ATOD and DTOA Expressions

The following pattern components can be used when outputting a date as a string. Of course, more than one component can be used in the same expression.

Note: You can add punctuation marks (e.g., dashes, slashes, commas) and spaces between pattern components as desired.

- **day** — weekday (Mon)
- **DD** — date in the month (15)
- **MM** — number of the month (01)
- **MMM** or **mmm** — abbreviated form (first three letters) of month name (Jan)
- **YY** — last two digits of year (06)
- **YYYY** — all four digits of year (2006)
- **hh:mm** — hours and minutes (12:05)

Converting a Date to a String: Examples

```
SELECT system.dbo.tabula_dtoa(demo.dbo.INVOICES.IVDATE,
'DD/MM/YY', ",")
FROM demo.dbo.INVOICES;

SELECT system.dbo.tabula_dtoa(demo.dbo.INVOICES.UDATE,
'DD-MM-YYYY hh:mm', ",")
FROM demo.dbo.INVOICES;

SELECT system.dbo.tabula_dtoa(demo.dbo.INVOICES.IVDATE,
'day DD MMM YYYY',
'Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct, Nov, Dec',
'Sun,Mon,Tue,Wed,Thu,Fri,Sat')
FROM demo.dbo.INVOICES;
```

Note: This function can be adapted to display dates in any language (e.g., Russian):

```
SELECT SYSTEM.DBO.TABULA_DTOA(DEMO.DBO.INVOICES.IVDATE,
'DAY DD MMM YYYY',
'IAN,FEV,MAR,APR,MAI,IUN,IUL,AVG,SEN,OKT,NOI,DEK',
'VSK,PND,VTR,SRD,CTV,PTN,SBT')
FROM DEMO.DBO.INVOICES
```

Converting a String to a Date: Examples

```
SELECT system.dbo.tabula_atod('01/06/2005', 'DD/MM/YYYY');
/* the integer corresponding to the date */

SELECT system.dbo.tabula_dateconvert(
system.dbo.tabula_atod('01/06/2005', 'DD/MM/YYYY'));
/* the date converted back to SQL format */
```

Chapter 17: Built-in ODBC Functions for Oracle Database

Introduction

Several ODBC functions have been written for ***Priority*** when using an Oracle database. This chapter contains a comprehensive list of string functions, number functions and date functions, together with examples of their usage.

String Functions

The following functions are performed on strings:

- **tabula.tabulaf.atoi** (*string*) — outputs the designated string as an integer.

  SELECT tabula.tabulaf.atoi('100') FROM DUAL; /* 100 */

- **tabula.tabulaf.ator** (*string*) — outputs the designated string as a real number.

  SELECT tabula.tabulaf.ator('100.66') FROM DUAL; /* 100.66 */

- **tabula.tabulaf.itoa** (*string, n*) — moves the decimal point *n* places to the left. When input is a string that contains one or more decimal points (which break up the string into parts), the function is applied separately to each part; this is useful for sorting columns that contain WBS codes.

  SELECT tabula.tabulaf.itoa('1',3) FROM DUAL; /* .001 */

  SELECT tabula.tabulaf.itoa('3.5.2',3) FROM DUAL; /* .003.005.002 */

- **tabula.tabulaf.strpiece** (*string, delimiter, m, n*) — for a given input string and delimiter (which breaks up the string into parts), retrieves *n* parts, beginning from the *m*th part.

  SELECT tabula.tabulaf.strpiece('hello-world','-',2,1) FROM DUAL; /* 'world' */

  SELECT tabula.tabulaf.strpiece('112.113.114.115','.',1,3) FROM DUAL;

  /* '112.113.114' */

Note: The following function is intended for users of ***Priority*** in Hebrew.

The TABULAF.HEBCONVERT function performs automated conversion of Hebrew text formats in strings that contain both English and Hebrew letters.

Example: The following command yields a string that contains both English and Hebrew letters. Note that the English letters and the numerals appear backwards.
SELECT F.DES FROM tabula.demo$FNCCONST F WHERE F.NAME2 = 'AutoRecon';
Result: 3/חשבוניות=2/הזמנות=1(בקבלה .אוט התאמה=OFIF(

Using the TABULAF.HEBCONVERT function, the output appears as follows:
SELECT tabula.tabulaf.hebconvert(tabula.tabulaf.revstr(F.DES)) FROM tabula.demo$FNCCONST F WHERE F.NAME2 = 'AutoRecon';
Result: (FIFO=3/חשבוניות=2/הזמנות=1) בקבלה .אוט התאמה

Number Functions

The following functions are performed on numbers:

- **tabula.tabulaf.tabula_htoi**(*m*) — inputs a hexadecimal value and translates it to its corresponding integer.

  SELECT tabula.tabulaf.htoi('ff') FROM DUAL; /* 255 */

- **tabula.tabulaf.tabula_itoh**(*n*) — returns the hexadecimal value of *n*, where *n* is an integer.

  SELECT tabula.tabulaf.itoh(15) FROM DUAL; /* 'f' */

Shifted Integers

When working with shifted integers, there is often a need to convert them to real numbers. In ***Priority***, the REALQUANT function performs this conversion automatically.

There is currently no ODBC-compliant equivalent to the REALQUANT function. Therefore, when writing queries using SQL tools, it is necessary to execute the division manually. The number of places that the decimal point should be moved is determined by the value of the DECIMAL system constant (usually, 3).

Example:
SELECT TQUANT, TQUANT / 1000
FROM TABULA.DEMO$ORDERITEMS WHERE ORD <> 0
/* assuming the DECIMAL constant = 3 */

Notes:

If the decimal precision is defined as 2 or 1, the integer will need to be divided by 100 or 10, accordingly.

To check the value of the DECIMAL system constant, execute the following query:
SELECT VALUE FROM TABULA.SYSCONST WHERE NAME = 'DECIMAL';

Date Functions

In ***Priority***, dates, times and days are stored in the database as integers, which correspond to the number of minutes elapsed since Jan. 1, 1988 (for example, Dec. 31, 1987 = -1440). These integers can be retrieved using the *SQL Development* (**WINDBI**) program by writing a query such as:

```
SELECT 0 + IVDATE FROM INVOICES FORMAT;
```

This enables you to perform calculations on dates.

Example: To calculate the day after an invoice date, use:
SELECT IVDATE + 24:00 FROM INVOICES FORMAT;

The following functions are performed on dates:

- **tabula.tabulaf.bofyear** (*date*) — yields the date of the first day of the year.

  ```
  SELECT tabula.tabulaf.bofyear(tabula.demo$INVOICES.IVDATE)
  FROM tabula.demo$INVOICES;
  ```

- **tabula.tabulaf.eofyear** (*date*) — yields the date of the end of the year.

  ```
  SELECT tabula.tabulaf.eofyear(tabula.demo$INVOICES.IVDATE)
  FROM tabula.demo$INVOICES;
  ```

- **tabula.tabulaf.bofhalf** (*date*) yields the date of the first day of the six-month period (half a year) in which the date falls.

  ```
  SELECT tabula.tabulaf.bofhalf(tabula.demo$INVOICES.IVDATE)
  FROM tabula.demo$INVOICES;
  ```

- **tabula.tabulaf.eofhalf** (*date*) — yields the date of the end of the half-year.

  ```
  SELECT tabula.tabulaf.eofhalf(tabula.demo$INVOICES.IVDATE)
  FROM tabula.demo$INVOICES;
  ```

- **tabula.tabulaf.bofquarter** (*date*) — yields the date of the first day of the quarter.

  ```
  SELECT tabula.tabulaf.bofquarter(tabula.demo$INVOICES.IVDATE)
  FROM tabula.demo$INVOICES;
  ```

- **tabula.tabulaf.eofquarter** (*date*) — yields the date of the end of the quarter.

  ```
  SELECT tabula.tabulaf.eofquarter(tabula.demo$INVOICES.IVDATE)
  FROM tabula.demo$INVOICES;
  ```

- **tabula.tabulaf.bofmonth** (*date*) — yields the date of the first day of the month.

  ```
  SELECT tabula.tabulaf.bofmonth(tabula.demo$INVOICES.IVDATE)
  FROM tabula.demo$INVOICES;
  ```

- **tabula.tabulaf.eofmonth** (*date*) — yields the date of the end of the month.

  ```
  SELECT tabula.tabulaf.eofmonth(tabula.demo$INVOICES.IVDATE)
  FROM tabula.demo$INVOICES;
  ```

- **tabula.tabulaf.get_week** (*date, day year began*) — yields an integer comprised of the year (last two digits of the year) and the number of the week in the year (two digits, between 01 and 53).

  SELECT tabula.tabulaf.get_week(tabula.demo$INVOICES.IVDATE,3) FROM tabula.demo$INVOICES;

- **tabula.tabulaf.bofweek** (*week no., day year began*) — given a value for a week (the last two digits of a year and the number of a week in that year) and the day the year started, yields the date that week started.

  SELECT tabula.tabulaf.bofweek(0801,1) FROM DUAL;

- **tabula.tabulaf.get_week6** (*date, day year began*) — yields an integer comprised of the year in 4 digits and the number of the week in the year (two digits, between 01 and 53).

  SELECT tabula.tabulaf.get_week6(tabula.demo$INVOICES.IVDATE,3) FROM tabula.demo$INVOICES;

- **tabula.tabulaf.mweek** (*week, day year began*) — given a value for a week (the last two digits of a year and the number of a week in that year) and the day the year began, yields the number of the month in which that week falls.

  SELECT tabula.tabulaf.mweek(0819,3) FROM DUAL;

- **tabula.tabulaf.dtoa** (*date, pattern, x, y)* — converts a date to a string. If you are not displaying the names of the days or months, the last two parameters should be empty strings.

  See usage below.

- **tabula.tabulaf.atod** (*string, pattern, x, y*) — converts a string to a date.

  See usage below.

Date Pattern Components for ATOD and DTOA Expressions

The following pattern components can be used when outputting a date as a string. Of course, more than one component can be used in the same expression.

Note: You can add punctuation marks (e.g., dashes, slashes, commas) and spaces between pattern components as desired.

- **DD** — date in the month (15)
- **MM** — number of the month (01)
- **YY** — last two digits of year (06)
- **YYYY** — all four digits of year (2006)
- **hh:mm** — hours and minutes (12:05)

Converting a Date to a String: Examples

SELECT tabula.tabulaf.dtoa(IVDATE, 'DD/MM/YY', 0, 0) FROM tabula.demo$INVOICES;

```
SELECT tabula.tabulaf.dtoa(UDATE, ' DD-MM-YYYY hh:mm', 0, 0)
FROM tabula.demo$INVOICES;
```

Converting a String to a Date: Example

```
SELECT tabula.tabulaf. atod('01/06/2005', 'DD/MM/YYYY') FROM
DUAL;
/* the integer corresponding to the date */
```