ORACLE
PeopleSoft

PeopleTools 8.58: SQR for PeopleSoft Developers

May 2020

ORACLE

PeopleTools 8.58: SQR for PeopleSoft Developers

Contents

Preface: Preface ..... **ix**
Understanding the PeopleSoft Online Help and PeopleBooks ix
Hosted PeopleSoft Online Help ix
Locally Installed Help ix
Downloadable PeopleBook PDF Files ix
Common Help Documentation ix
Field and Control Definitions x
Typographical Conventions x
ISO Country and Currency Codes x
Region and Industry Identifiers xi
Translations and Embedded Help xi
Using and Managing the PeopleSoft Online Help xii
PeopleTools Related Links xii
Contact Us xii
Follow Us xii
Chapter 1: Getting Started with SQR for PeopleSoft **15**
SQR for PeopleSoft Overview 15
SQR for PeopleSoft Implementation 15
Other Sources of Information 16
Chapter 2: Introducing a Sample Structured Query Report Program **17**
Using This Guide 17
Setting Up the Sample Database 19
Considerations for DBX 20
Understanding the Sample Program for Printing a Text String 20
Creating and Running a Sample SQR Program 20
Creating an SQR Program 21
Running an SQR Program 21
Viewing SQR Output 22
Chapter 3: Creating Headings and Footings **23**
Understanding SQR Pages 23
Creating Page Headings and Footings 23
Understanding the Heading and Footing Code Example 23
Adding Page Headings 24
Adding Page Footings 24
Chapter 4: Selecting Data from the Database **27**
Understanding the Sample Program for Listing and Printing Data 27
Creating SQR Select Paragraphs 28
Chapter 5: Using Column Variables **31**
Using Column Variables in Conditions 31
Changing Column Variable Names 31
Chapter 6: Using Break Logic **33**
Understanding Break Logic 33
Using the ON-BREAK Option 34
Skipping Lines Between Groups 35
Arranging Multiple Break Columns 35
Using Break Processing Enhancements 36

Controlling Page Breaks and Calculating Subtotals and Totals....36
Handling Page Breaks....38
Printing the Date....38
Obtaining Totals....39
Using Hyphens and Underscores....40
Setting Break Procedures with BEFORE and AFTER Qualifiers....40
Controlling Page Breaks with Multiple ON-BREAK Columns....43
Saving a Value When a Break Occurs....44
Using ON-BREAK on a Hidden Column....44
Performing Break Processing on Numeric Values....46
Chapter 7: Adding Declarations Using the SETUP Section....47
Understanding the SETUP Section....47
Creating a SETUP Section....47
Using the DECLARE-LAYOUT Command....48
Sample SETUP Program....48
Defining the SQR Page Layout....48
Overriding Default Settings....49
Declaring a Page Orientation....49
Chapter 8: Creating Master and Detail Reports....51
Understanding Master and Detail Reports....51
Understanding the Sample Program for Master and Detail Reports....51
Correlating Subqueries....52
Sample Program Output....53
Chapter 9: Creating Cross-Tabular Reports....55
Understanding Cross-Tabular Reports....55
Using an Array....56
Creating an Array....58
Grouping by Category....58
Using Multiple Arrays....60
Chapter 10: Printing Mailing Labels....63
Understanding Mailing Label Printing....63
Understanding the Sample Program for Printing Mailing Labels....63
Defining Columns and Rows....64
Running the Print Mailing Labels Program....65
Chapter 11: Creating Form Letters....67
DOCUMENT Paragraph....67
Sample Program for Form Letters....67
Chapter 12: Exporting Data to Other Applications....69
Understanding the Sample Program for Exporting Data....69
Creating an Export File....70
Chapter 13: Using Graphics....71
Understanding the Sample Program for Simple Tabular Reports....71
Adding Graphics....72
Sharing Images Among Reports....74
Printing Bar Codes....76
Chapter 14: Using Business Charts....77
Understanding Business Charts....77
Creating a Chart....77
Defining Charts....80
Printing Charts....80
Running the Program to Create Graphical Reports....81

Passing Data to Charts....81
Chapter 15: Changing Fonts....83
Setting Fonts....83
Positioning Text....83
Using the WRAP Option....85
Chapter 16: Writing Printer-Independent Reports....87
Understanding Printer-Independent Reports....87
Reviewing the Sample Program for Selecting the Printer Type at Runtime....88
Chapter 17: Using Dynamic SQL and Error Checking....89
Using Variables in SQL....89
Using Dynamic SQL....90
Using SQL Error Checking....91
Using SQL and Substitution Variables....92
Chapter 18: Using Procedures and Local Variables and Passing Arguments....95
Using Procedures....95
Using Local Variables....95
Passing Arguments....96
Chapter 19: Creating Multiple Reports from One Program....101
Understanding How to Create Multiple Reports....101
Understanding the Sample Program for Multiple Reports....101
Defining Heading and Footing Sections....103
Defining Program Output....104
Chapter 20: Using Additional SQL Statements with SQR....105
Using SQL Statements in SQR....105
Using the BEGIN-SQL Paragraph....105
Chapter 21: Working with Dates....107
Understanding Dates and Date Arithmetic....107
Using Literal Date Formats....109
Using String-to-Date Conversions....109
Using Date-to-String Conversions....110
Using Dates with the INPUT Command....110
Using Date Edit Masks....110
Declaring Date Variables....112
Chapter 22: Using National Language Support....113
Understanding Locales....113
Selecting Locales....113
Defining a Default Locale....114
Switching Locales....115
Modifying Locale Preferences....115
Specifying NUMBER, MONEY, and DATE Keywords....115
Chapter 23: Using Interoperability Features....117
Calling SQR from Another Application....117
Invoking an SQR Program by Using the SQR API....117
Invoking an External Application API by Using the UFUNC.C Interface....119
Adding a User Function....120
Understanding the UFUNC.C File....120
Adding a Function Prototype....120
Adding an Entry to the USERFUNCS Table....121
Adding Implementation Code....121
Relinking SQR....122
Using UFUNC in Microsoft Windows....123

Implementing New User Functions in Microsoft Windows....123
Chapter 24: Testing and Debugging....125
Using the Test Feature....125
Using the #DEBUG Command....125
Using Compiler Directives for Debugging....126
Avoiding Common Programming Errors....127
Chapter 25: Increasing Performance and Tuning....129
Understanding SQR Performance and SQL Statements....129
Simplifying Complex Select Paragraphs....129
Using LOAD-LOOKUP to Simplify Joins....130
Improving SQL Performance with Dynamic SQL....131
Examining SQL Cursor Status....132
Avoiding Temporary Database Tables....132
Understanding Temporary Database Tables....133
Using and Sorting Arrays....133
Using and Sorting Flat Files....136
Creating Multiple Reports in One Pass....138
Tuning SQR Numerics....138
Compiling SQR Programs and Using SQR Execute....139
Setting Processing Limits....139
Buffering Fetched Rows....140
Running Programs on the Database Server....140
Chapter 26: Compiling Programs and Using SQR Execute....141
Understanding Compile Features....141
Compiling and Running an SQR Program....141
Chapter 27: Printing with SQR....143
Specifying Output File Types by Using SQR Command-Line Flags....143
Using the DECLARE-PRINTER Command....144
Chapter 28: Using the SQR Command Line....147
Understanding the SQR Command Line....147
Specifying Command-Line Arguments....148
Understanding Command-Line Arguments....148
Retrieving Arguments....148
Specifying Arguments and Argument Files....149
Using an Argument File....149
Using Other Approaches to Pass Command-Line Arguments....150
Using Reserved Characters....150
Creating an Argument File from a Report....150
Using Batch Mode....151
Chapter 29: Generating and Publishing HTML from an SQR Program....153
Understanding SQR Capabilities That Are Available with HTML....153
Generating HTML Output....153
Understanding HTML Output....154
Producing HTML Output....154
Using -PRINTER:EH....155
Setting HTML Attributes Under -PRINTER:EH....156
Using -PRINTER:HT....158
Bursting Reports....158
Setting Attributes with HTML Procedures....159
Using Additional HTML Procedures....159
Setting Output File Types....160

Testing HTML Output....160
Using HTML Procedures in an SQR Program....160
Understanding HTML Procedures....161
Using HTML Procedures....161
Positioning Objects....161
Displaying Records in Tables....162
Creating Headings....163
Highlighting Text....163
Creating Links....164
Including Images....165
Displaying Text in Lists....165
Formatting Paragraphs....166
Incorporating Your Own HTML Tags....166
Modifying an Existing SQR Program for HTML....167
Publishing a Report....168
Publishing Reports....168
Supporting Older Browsers....169
Viewing Published Reports....169
Publishing by Using an Automated Process....169
Publishing by Using a CGI Script....170
Chapter 30: Generating Tagged PDF Output from SQR Program....173
Tagged PDF Overview....173
Sample Program to Create Tagged PDF....173
SQR Program for using Paragraph in tagged content....173
Generating a Tagged Table in a PDF Report....173
Generating a Tagged List in a PDF Report....174
Generating Alternate Text for a Figure....175
Tagged PDF in PeopleSoft Application....175
Using Accessibility Checkers....175
Chapter 31: Generating XML Output from SQR Program....177
Generating XML Output....177
SQR Commands to Generate XML Output....177
Sample Program to Generate XML Output....178
Generating XML Output in PeopleSoft Applications....181
Chapter 32: Creating a Table of Contents....183
Using the DECLARE-TOC Command....183
Using the TOC-ENTRY Command....184
Adding a Table of Contents to the CUST.SQR Sample Program....184

Preface

Understanding the PeopleSoft Online Help and PeopleBooks

The PeopleSoft Online Help is a website that enables you to view all help content for PeopleSoft applications and PeopleTools. The help provides standard navigation and full-text searching, as well as context-sensitive online help for PeopleSoft users.

Hosted PeopleSoft Online Help

You can access the hosted PeopleSoft Online Help on the Oracle Help Center. The hosted PeopleSoft Online Help is updated on a regular schedule, ensuring that you have access to the most current documentation. This reduces the need to view separate documentation posts for application maintenance on My Oracle Support. The hosted PeopleSoft Online Help is available in English only.

To configure the context-sensitive help for your PeopleSoft applications to use the Oracle Help Center, see Configuring Context-Sensitive Help Using the Hosted Online Help Website.

Locally Installed Help

If you're setting up an on-premise PeopleSoft environment, and your organization has firewall restrictions that prevent you from using the hosted PeopleSoft Online Help, you can install the online help locally. See Configuring Context-Sensitive Help Using a Locally Installed Online Help Website.

Downloadable PeopleBook PDF Files

You can access downloadable PDF versions of the help content in the traditional PeopleBook format on the Oracle Help Center. The content in the PeopleBook PDFs is the same as the content in the PeopleSoft Online Help, but it has a different structure and it does not include the interactive navigation features that are available in the online help.

Common Help Documentation

Common help documentation contains information that applies to multiple applications. The two main types of common help are:

- Application Fundamentals
- Using PeopleSoft Applications

Most product families provide a set of application fundamentals help topics that discuss essential information about the setup and design of your system. This information applies to many or all applications in the PeopleSoft product family. Whether you are implementing a single application, some combination of applications within the product family, or the entire product family, you should be familiar with the contents of the appropriate application fundamentals help. They provide the starting points for fundamental implementation tasks.

In addition, the *PeopleTools: Applications User's Guide* introduces you to the various elements of the PeopleSoft Pure Internet Architecture. It also explains how to use the navigational hierarchy, components, and pages to perform basic functions as you navigate through the system. While your application or implementation may differ, the topics in this user's guide provide general information about using PeopleSoft applications.

Field and Control Definitions

PeopleSoft documentation includes definitions for most fields and controls that appear on application pages. These definitions describe how to use a field or control, where populated values come from, the effects of selecting certain values, and so on. If a field or control is not defined, then it either requires no additional explanation or is documented in a common elements section earlier in the documentation. For example, the Date field rarely requires additional explanation and may not be defined in the documentation for some pages.

Typographical Conventions

The following table describes the typographical conventions that are used in the online help.

Typographical Convention	***Description***
Key+Key	Indicates a key combination action. For example, a plus sign (+) between keys means that you must hold down the first key while you press the second key. For Alt+W, hold down the Alt key while you press the W key.
. . . (ellipses)	Indicate that the preceding item or series can be repeated any number of times in PeopleCode syntax.
{ } (curly braces)	Indicate a choice between two options in PeopleCode syntax. Options are separated by a pipe (|).
[ ] (square brackets)	Indicate optional items in PeopleCode syntax.
& (ampersand)	When placed before a parameter in PeopleCode syntax, an ampersand indicates that the parameter is an already instantiated object. Ampersands also precede all PeopleCode variables.
⇒	This continuation character has been inserted at the end of a line of code that has been wrapped at the page margin. The code should be viewed or entered as a single, continuous line of code without the continuation character.

ISO Country and Currency Codes

PeopleSoft Online Help topics use International Organization for Standardization (ISO) country and currency codes to identify country-specific information and monetary amounts.

ISO country codes may appear as country identifiers, and ISO currency codes may appear as currency identifiers in your PeopleSoft documentation. Reference to an ISO country code in your documentation

does not imply that your application includes every ISO country code. The following example is a country-specific heading: "(FRA) Hiring an Employee."

The PeopleSoft Currency Code table (CURRENCY_CD_TBL) contains sample currency code data. The Currency Code table is based on ISO Standard 4217, "Codes for the representation of currencies," and also relies on ISO country codes in the Country table (COUNTRY_TBL). The navigation to the pages where you maintain currency code and country information depends on which PeopleSoft applications you are using. To access the pages for maintaining the Currency Code and Country tables, consult the online help for your applications for more information.

Region and Industry Identifiers

Information that applies only to a specific region or industry is preceded by a standard identifier in parentheses. This identifier typically appears at the beginning of a section heading, but it may also appear at the beginning of a note or other text.

Example of a region-specific heading: "(Latin America) Setting Up Depreciation"

Region Identifiers

Regions are identified by the region name. The following region identifiers may appear in the PeopleSoft Online Help:

- Asia Pacific
- Europe
- Latin America
- North America

Industry Identifiers

Industries are identified by the industry name or by an abbreviation for that industry. The following industry identifiers may appear in the PeopleSoft Online Help:

- USF (U.S. Federal)
- E&G (Education and Government)

Translations and Embedded Help

PeopleSoft 9.2 software applications include translated embedded help. With the 9.2 release, PeopleSoft aligns with the other Oracle applications by focusing our translation efforts on embedded help. We are not planning to translate our traditional online help and PeopleBooks documentation. Instead we offer very direct translated help at crucial spots within our application through our embedded help widgets. Additionally, we have a one-to-one mapping of application and help translations, meaning that the software and embedded help translation footprint is identical—something we were never able to accomplish in the past.

Using and Managing the PeopleSoft Online Help

Select About This Help in the left navigation panel on any page in the PeopleSoft Online Help to see information on the following topics:

- Using the PeopleSoft Online Help
- Managing Hosted online help
- Managing locally installed PeopleSoft Online Help

PeopleTools Related Links

PeopleTools 8.58 Home Page

PeopleTools Elasticsearch Home Page

"PeopleTools Product/Feature PeopleBook Index" (PeopleTools 8.58: Getting Started with PeopleTools)

PeopleSoft Hosted Online Help

PeopleSoft Information Portal

PeopleSoft Spotlight Series

PeopleSoft Training and Certification | Oracle University

My Oracle Support

Oracle Help Center

Contact Us

Send your suggestions to psoft-infodev_us@oracle.com. Please include the applications update image or PeopleTools release that you're using.

Follow Us

Facebook.

YouTube

Twitter@PeopleSoft_Info.

PeopleSoft Blogs

LinkedIn

Chapter 1

Getting Started with SQR for PeopleSoft

SQR for PeopleSoft Overview

SQR for PeopleSoft is both a language and a set of tools that enables you to create professional reports:

- SQR is a programming language for accessing and manipulating data to create custom reports. SQR has many advantages, including portability across multiple platforms and relational database management systems and support of SQL data manipulation capabilities. It is also a fourth-generation language; it is closer to human languages and, therefore, is more intuitive than first-, second-, or third-generation languages. SQR for PeopleSoft enables you to design report layouts, generate a variety of output types—including complex tabular reports, multiple page reports, form letters, mailing labels, and more—and create HTML, PDF, XML, or configured output for laser printers and phototypesetters.
- SQR Execute enables you to run previously compiled SQR programs.
- SQR Print enables you to configure reports for most printers.
- SQR also provides a library of sample programs and output that you can use both as a learning tool and as a basis for creating your own reports. These samples reside in the SQR for PeopleSoft directory <PS_HOME>\bin\sqr\<database_platform>\SAMPLE (or SAMPLEW, for Windows).

See SQR Language Reference for PeopleSoft.

SQR for PeopleSoft Implementation

This section describes the prerequisites for implementing SQR for PeopleSoft.

- You need a sound understanding of SQL and structured programming languages to use the SQR language.
- You do not need to carry out a separate installation procedure because SQR for PeopleSoft is installed automatically when you install PeopleTools.
- Typically, you should use Application Engine to run background SQL processing programs. You may want to explore whether Application Engine can meet your needs before delving into SQR.

  See Application Engine.
- You can run SQR programs locally by using the SQR executable (for Microsoft Windows it is SQRW) and through the PeopleSoft Process Scheduler. For details on installing Process Scheduler and running SQRs using Process Scheduler,

See "PeopleSoft Process Scheduler" (PeopleTools 8.58: Process Scheduler) and "Understanding the Management of PeopleSoft Process Scheduler" (PeopleTools 8.58: Process Scheduler).

See the product documentation for *PeopleSoft 9.2 Application Installation*.

Note: From PeopleTools 8.54 release, the SQR executables, such as, SQRW,SQWT,SQRWP, and SQRWT are x64 bit binary on Windows platforms.

Other Sources of Information

This section provides information to consider before you begin to use SQR for PeopleSoft.

In addition to implementation considerations presented in this section, take advantage of all PeopleSoft sources of information, including the installation guides, release notes, PeopleBooks, red papers, the Updates + Fixes area of My Oracle Support, and the PeopleSoft curriculum courses.

Chapter 2

Introducing a Sample Structured Query Report Program

Using This Guide

Initial sections of this guide teach the basic uses of SQR. You learn how to:

- Create a variety of reports, such as tabular, cross-tabular, and master and detail reports.
- Produce mailing labels, form letters, and envelopes.
- Enhance your reports with typeset-quality fonts and graphics.
- Produce graphs and charts that help you present data and trends visually.

Subsequent sections describe the advanced features and uses of SQR. You learn how to:

- Create HTML output and publish reports on the internet, an intranet, or an extranet.
- Create reports that can be easily ported between different systems and databases and that support different printer and display types.
- Create reports that format dates, numbers, and money according to local preferences.
- Integrate SQR with other software packages, such as front-end user interface tools and spreadsheets.
- Extend SQR with procedures and functions that are written in C.
- Test and debug programs.
- Tune programs for optimum performance.

The code examples demonstrate standard SQR programming style. Use this standard style to make your code easier for other SQR programmers to understand.

You can run the program examples in this guide without modification against the Oracle database, and you can run them against other databases with minor modifications.

Audience

This guide was written for programmers who develop reports for relational databases. To use this guide effectively, you need a working knowledge of SQL and experience writing software programs. You also must be familiar with your particular database and operating system.

How to Use SQR for PeopleSoft Developers

You can just read this book and study the sample programs. However, Oracle encourages you to try these programs for yourself and to experiment with them. Make some changes to the sample programs and see how they run.

To use the sample programs, you must first install SQR for PeopleSoft. SQR for PeopleSoft installs automatically when you install PeopleTools.

If you installed all of the program components, the sample programs are located in the TUTORIAL directory underneath <PS_HOME>\bin\sqr\<database_platform>.

You can run the sample programs on any hardware platform, but you may find it easier to review SQR program results from the Microsoft Windows platform by using the SQR Viewer or a web browser to verify your results.

Note: You can set up the sample database and run the sample programs with any username and password, although you may want to use an account that does not hold important data.

Related Documents

In addition to this developer's guide, SQR for PeopleSoft includes *SQR for PeopleSoft Language Reference,* a complete reference to SQR commands, arguments, and command-line flags.

For information about supported database platforms, see Supported Platforms on My Oracle Support. You can also consult the *PeopleTools Hardware and Software Requirements* guide for a snapshot of current requirements.

Syntax Conventions

Syntax and code examples use the following conventions:

Convention	***Description***
{ }	Braces enclose required items.
[ ]	Square brackets enclose optional items.
...	Ellipses indicate that the preceding parameter can be repeated.
|	A vertical bar separates alternatives within brackets, braces, or parentheses.
'	A single quote starts and ends a literal text constant or any argument that has more than one word. **Important!** If you are copying code directly from the examples in the PDF file, make sure that you change the slanted quotes to regular quotes; otherwise, you will receive an error message.
,	A comma separates multiple arguments.

Convention	*Description*
( )	Parentheses must enclose an argument or element.
UPPERCASE	SQR commands and arguments are uppercase within the text but lowercase in the code examples. (Note that these commands are *not* case-sensitive.)
Variable	Information and values that you must supply appear in *variable* style.
hyphen versus underscore	Many SQR commands, such as BEGIN-PROGRAM, use a hyphen, whereas procedure and variable names use an underscore. Procedure and variable names can contain either a hyphen or underscores, but using underscores in procedure and variable names to distinguish them from SQR commands is best. It also prevents confusion when you mix variable names and numbers in an expression, where hyphens could be mistaken for minus signs.

Setting Up the Sample Database

To run the sample programs in this guide, you must create a sample database. To do so, run the loadall.sqr program:

1. Change to the SAMPLE (or SAMPLEW for Microsoft Windows) directory under <PS_HOME>\bin \sqr\<database_platform>.
2. At the command line, enter:

```
sqr loadall username/password
```

If SQR is installed on Microsoft Windows, you can run loadall.sqr by double-clicking the Loadall icon. If your system does not display this icon, run loadall.sqr from the SAMPLEW directory of SQR for PeopleSoft.

If an individual table already exists, you are prompted to enter:

- *A*: Abort the load.
- *S*: Skip the specified table.
- *R*: Reload the specified table.
- *C*: Reload all tables.

You can also run this as a batch program by entering the preferred option (*A*, *S*, *R*, or *C*) at the command-line. For example:

```
sqr loadall username/password a
```


Considerations for DBX

The following considerations apply for DB2 on AIX and DB2 on z/OS.

DB2 on AIX

The DB2CLI.INI file (on Windows with the DB2 ODBC connection) should have the following entry.

This file is typically located in C:\Apps\DB\Db2 directory.

```
[common]PATCH2=6DISABLEKEYSETCURSOR=1
```

DB2 on z/OS

The PSSQR.UNX or PSSQR.INI file should have the following line:

```
FORCESPACEAFTERCOMMA=TRUE
```

Understanding the Sample Program for Printing a Text String

The first sample program is the simplest SQR program. It prints a text string:

```
Program ex1a.sqr
begin-program
   print 'Hello, World.' (1,1)
end-program
```

Note: For your convenience, all of the program examples and their output files are included with the installation. As mentioned, these samples are in the SQR for PeopleSoft directory <PS_HOME>\bin\sqr\<database_platform>\SAMPLE (SAMPLEW for Microsoft Windows).

Take another look at the sample program. This program contains three lines of code, starting with BEGIN-PROGRAM and ending with END-PROGRAM. These two commands and the code between them make up the PROGRAM section, which is used to control the order of processing. The PROGRAM section is required, and you can have only one. It typically goes at or near the top of the program.

The PROGRAM section contains a PRINT command, which in this case prints the text *Hello, World.* This text is enclosed in single quotation marks ('), which are used in SQR to distinguish literal text from other program elements.

The last element of the PRINT command indicates the position on the output page. An output page can be thought of as a grid of lines and columns. The pair (1,1) indicates line 1, column 1, which is the upper-left corner of the page.

Note: In SQR, you must place each command on a new line. You can indent SQR commands.

Creating and Running a Sample SQR Program

This section discusses how to:

- Create an SQR program.
- Run an SQR program.

Creating an SQR Program

To create an SQR program:

1. Open a text editor and enter the code in the sample program exactly as shown or open the ex1a.sqr file from the TUTORIAL directory.
2. If you are writing the sample program, save your code with the name ex1a.sqr.

   SQR programs usually have a file extension of .sqr.

Running an SQR Program

To run the sample program:

1. Change to the directory in which you saved the program using the command that is appropriate to your operating system.
2. Enter the appropriate SQR program command at the system command prompt (UNIX/Linux or Microsoft Windows) or from within the graphical user interface (GUI) of the SQR application, where available (Microsoft Windows only).

   If you are using the command line, use SQR (UNIX/Linux) or SQRW (Microsoft Windows) to invoke SQR. Enter sqr or sqrw, the SQR program name, and the connectivity string, all on one line, using this syntax:

   ```
   [sqr or sqrw] [program] [connectivity] [flags ...] [args ...] [@file ...]
   ```

In a common configuration, you may be running SQR on Microsoft Windows against an Oracle database that is located on another machine in the network. Use this command format:

```
sqrw ex1a username/password@servername -KEEP
```

If you correctly replace *username*, *password*, and *servername* with the appropriate information, you should have a command line like this:

```
sqrw ex1a sammy/baker@rome -KEEP
```

To produce the output file for this exercise, the example uses the -KEEP flag, which is defined later in this guide.

See "SQR for PeopleSoft Tools" (PeopleTools 8.58: SQR Language Reference for PeopleSoft).

See Specifying Output File Types by Using SQR Command-Line Flags.

Command Line Examples

Here are some examples for running SQR from the command line for different databases and platforms.

DB2 on Microsoft Windows

```
%PS_HOME%\bin\sqr\DB2\BINW\sqrw %PS_HOME%\sqr\xrfwin.sqr T846U10/testdb2/t3stdb20
-oc:\sqr_out\xrfwin.out -i%PS_HOME%\sqr\; -zif%PS_HOME%\sqr\pssqr.ini
```



```
-fc:\sqr_out\ T846U10 T846U10 952 VP1 testEnglish
```

Oracle on Unix

```
$PS_HOME/bin/sqr/ORA/bin/sqr $PS_HOME/sqr/xrfwin.sqr T846U22/T846U22@T846U22
-o$PS_HOME/xrfwin_689.out -i$PS_HOME/sqr/ -ZIF$PS_HOME/sqr/pssqr.unx
"-f$PS_HOME/xrfwin_689.pdf" -printer:pd T846U22 689 VP1 PJS
```

Microsoft SQL Server on Microsoft Windows

```
%PS_HOME%\bin\sqr\MSS\BINW\sqrw %PS_HOME%\sqr\xrfwin.sqr T846U10/testdb2/t3stdb20
-oc:\sqr_out\xrfwin.out -i%PS_HOME%\sqr\; -zif%PS_HOME%\sqr\pssqr.ini
-fc:\sqr_out\ T846U10 T846U10 952 VP1 testEnglish
```

Viewing SQR Output

SQR normally places the SQR program output files in the directory from which you run the program. The output file has the same file name as the SQR file that created it, but the file extension is different.

The output files should appear as soon as your program has finished running. If you specified the -KEEP argument, one output file is in SQR Portable Format (recognizable by its .spf extension). SQR Portable Format is discussed later in this guide, but for now, you can view the sample program .spf file output, <filename>.spf, on Microsoft Windows platforms with the SQR Viewer GUI (sometimes referred to as an SPF Viewer). Invoke the SQR Viewer by entering sqrw at the command line.

On Microsoft Windows and UNIX/Linux systems, the program also produces an output file with an .lis extension. You can view this output file type from the command line with such commands as TYPE on Microsoft Windows systems or CAT, MORE, and VI on UNIX/Linux systems. Use the command that is appropriate to your system to view or print the .lis file.

The output for the example program looks like this for all platforms:

```
Hello, World.
```

You may also see a character such as ^L or <FF> at the end of this output file. It is the form-feed character that ejects the last page. This guide does not show form-feed characters.

Chapter 3

Creating Headings and Footings

Understanding SQR Pages

Typically, every page of a report has some information about the report itself, such as the title, the date, and the page number. In SQR, the page can be subdivided into three logical areas:

- The top area of the page is the *heading,* which is where the report title and the date normally print.
- The middle part of the page is the *body,* which is where the report data prints.
- The bottom area of the page is the *footing,* which is where the page number normally prints.

The heading, body, and footing of a page each has independent line numbers. You can print in each of these page areas by using line numbers that are relative to the top corner of that area without being concerned about the size of the other areas. That is, you can print to the first line of the body by using line number 1, independent of the size of the heading.

Note: Any space that is reserved for the heading and footing is taken from the body area of the page. With one line each in the heading and footing, the maximum possible size of the body of the report is reduced by two lines. Note also that line 1 of the body is actually the first line after the heading.

Creating Page Headings and Footings

This section provides an overview of the heading and footing code example and discusses how to:

- Add page headings.
- Add page footings.

Understanding the Heading and Footing Code Example

Here is an example of the code that is required to add a page heading and footing to a program:

```
Program ex2a.sqr
begin-program
   print 'Hello, World.' (1,1)
end-program
begin-heading 1
   print 'Tutorial Report' (1) center
end-heading
begin-footing 1
   ! print "Page n of m" in the footing
   page-number (1,1) 'Page '
   last-page   () ' of '
end-footing
```

The output for the ex2a.sqr program is:

```
Tutorial Report
Hello, World.

Page 1 of 1
```

Note: The PRINT command places text in memory, not on paper. SQR for PeopleSoft always prepares a page in memory before printing it to paper, creating the body first and then the HEADING and FOOTING sections. In this example, *Hello, World* is run first, followed by *Tutorial Report* and *Page 1 of 1*.

Adding Page Headings

Define a page heading in the HEADING section. Begin the section with a BEGIN-HEADING command and end it with an END-HEADING command. Follow the BEGIN-HEADING command with a number that represents the number of lines that are reserved for the heading. (In this example, *1* indicates a heading of one line.)

In the heading and footing sample program, the heading uses exactly one line and contains the text *Tutorial Report.* The CENTER argument ensures that the text is centered on the line.

Adding Page Footings

Define the page footing in the FOOTING section. Begin the section with a BEGIN-FOOTING command and end it with an END-FOOTING command. Follow the BEGIN-FOOTING command with a number that represents the number of lines that are reserved for the footing. (In this example, the *1* indicates a footing of one line.) This line consists of the text *Page 1 of 1*.

Adding Comments

Precede comments with an exclamation mark. The comment extends from the exclamation mark to the end of the line.

In the heading and footing sample program, the first line in the FOOTING section is a comment.

To print an exclamation mark , enter it twice to indicate that it is not the beginning of a comment. For example:

```
print 'Hello, World!!' (1,1)
```

Adding Page Numbers

Use the PAGE-NUMBER command to print the text *Page* and the current page number. Use the LAST-PAGE command to print the number of the last page, preceded by the word *of,* which is bracketed by spaces.

In the headings and footings code example, *Page 1 of 1* appears because only one page exists.

Indicating the Print Position

Include numbers in parentheses following the PRINT, PAGE-NUMBER, and LAST-PAGE commands to indicate the position for printing. Express a position in SQR language with three numbers in parentheses: line number, column number (character position), and width of the text.

In many cases, a position contains only the line and column numbers. The width is normally omitted because it is set by default to the width of the text that is being printed. If you also omit the line and column numbers, then the print position is set by default to the current position, which is the position following the last printed item.

In the heading and footing sample program, the LAST-PAGE command has the position (), so the current position is the position following the page number.

The print position is a point within the area of the page or, more precisely, within the heading, body, or footing. The position (1,1) in the heading is not the same as the position (1,1) in the body. Line 1 of the body is the first line following the heading. In the program, the heading has only one line, so line 1 of the body is actually the second line of the page. Similarly, line 1 of the footing is at the bottom of the page. It is the first line following the body.

Chapter 4

Selecting Data from the Database

Understanding the Sample Program for Listing and Printing Data

Here is a sample program that selects data from the database and prints it in columns:

```
Program ex3a.sqr
begin-program
   do list_customers
end-program
begin-heading 4
   print 'Customer Listing' (1) center
   print 'Name' (3,1)
   print 'City' (,32)
   print 'State' (,49)
   print 'Phone' (,55)
end-heading
begin-footing 1
   ! Print "Page n of m" in the footing
   page-number (1,1) 'Page '
   last-page   () ' of '
end-footing
begin-procedure list_customers
begin-select
name (,1)
city (,32)
state (,49)
phone (,55)
   position (+1)  ! Advance to the next line
from customers
end-select
end-procedure ! list_customers
```

The output for the ex3a.sqr program is:

```
                      Customer Listing

Name                           City           State  Phone
Gregory Stonehaven             Everretsville   OH    2165553109
John Conway                    New York        NY    2125552311
Eliot Richards                 Queens          NY    2125554285
Isaiah J Schwartz and Company  Zanesville      OH    5185559813
Harold Alexander Fink          Davenport       IN    3015553645
Harriet Bailey                 Mamaroneck      NY    9145550144
Clair Butterfield              Teaneck         NJ    2015559901
Quentin Fields                 Cleveland       OH    2165553341
Jerry's Junkyard Specialties   Frogline        NH    6125552877
Kate's Out of Date Dress Shop  New York        NY    2125559000
Sam Johnson                    Bell Harbor     MI    3135556732
Joe Smith and Company          Big Falls       NM    8085552124
Corks and Bottles, Inc.        New York        NY    2125550021
Harry's Landmark Diner         Miningville     IN    3175550948

Page 1 of 1
```

The PROGRAM section contains a single DO command, which invokes the **list_customers** procedure.

In SQR language, a procedure is a group of commands that is performed one after the other, as a procedure (or subroutine) is in other programming languages. A DO command invokes a procedure.

Break your program logic into procedures and keep the PROGRAM section small. It should normally contain a few DO commands for the main components of your report.

The HEADING section creates headings for the report columns. In this example, four lines are reserved for the heading:

```
begin-heading 4
   print 'Customer Listing' (1) center
   print 'Name' (3,1)
   print 'City' (,32)
   print 'State' (,49)
   print 'Phone' (,55)
end-heading
```

The Customer Listing title is printed on line 1. Line 2 is left blank. The first column heading, Name, is positioned at line 3 of the heading, in character position 1. The rest of the column heading commands omit the line numbers in their positions and are set by default to the current line. Line 4 of the heading is left blank.

In this sample program, the footing is the same as the one in the previous sample program.

Creating SQR Select Paragraphs

The BEGIN-SELECT command is the principal method of retrieving data from the database and printing it in a report. Look again at the sample program for listing and printing data, in which the **list_customers** procedure starts with BEGIN-PROCEDURE and ends with END-PROCEDURE.

Note the comment following the END-PROCEDURE command. It indicates that the procedure is being ended, which is helpful when you have a program with many procedures. (You can also omit the exclamation point, for example, END-PROCEDURE main.)

The procedure itself contains a select paragraph, which starts with BEGIN-SELECT and ends with END-SELECT.

The select paragraph is unique. It combines a SQL SELECT statement with SQR processing in a seamless way. The actual SQL statement is:

```
SELECT NAME, CITY, STATE, PHONE
FROM CUSTOMERS
```

Syntax of the Select Paragraph

In an SQR select paragraph, the SQL statement SELECT is omitted, and no commas are between the column names. Instead, each column is on its own line. You can also place SQR commands between the column names, and these commands are run for every record that the select fetches.

Note: You must name each individual column in a table. The SQL `SELECT * FROM` statement is not allowed in SQR.

SQR distinguishes column names from SQR commands in a select paragraph by their indention. You must place column names at the beginning of a line. You must indent SQR commands at least one space. In the following example, the POSITION command is indented to prevent it from being taken as a column name. The word *From* must be the first word in a line. The rest of the SQR select paragraph is then written freely, after SQL syntax.

Think of the select paragraph as a loop. The SQR commands, including printing of columns, are run in a loop, once for each record that Select returns. The loop ends after the last record is returned.

Data Positioning

In a select paragraph, you see positioning after each column name. This positioning implies a PRINT command for that column. Omitting the line number in the position causes it to be set by default to the current line.

```
begin-select
name (,1)
city (,32)
state (,49)
phone (,55)
   position (+1)  ! Advance to the next line
from customers
end-select
```

The implied PRINT command is a special SQR feature that is designed to save you coding time. It works only inside a select paragraph.

After the last column is a POSITION command: POSITION(+1). The plus sign (or minus sign) indicates relative positioning in SQR. A plus sign moves the print position forward from the current position, and a minus sign moves it back. The +1 in the sample program specifies one line down from the current line. This command advances the current print position to the next line.

Note: When you indicate print positions by using plus or minus signs, be sure that your numbers do not specify a position outside of the page boundaries.

Chapter 5

Using Column Variables

Using Column Variables in Conditions

You can name database columns with variables and use their values in conditions and commands.

When you select columns from the database in a select paragraph, you can immediately print them by using a position. For example:

```
begin-select
phone (,1)
   position (+1)
from customers
end-select
```

This example shows how to use the value of *phone* for another purpose, for example, in a condition:

```
begin-program
   do list_customers
end-program
begin-procedure list_customers
begin-select
phone
   if &phone = ''
      print 'No phone' (,1)
   else
      print &phone (,1)
   end-if
   position (+1)
from customers
end-select
end-procedure ! list_customers
```

The phone column is a SQR column variable. Precede column variables with an ampersand (&).

Unlike other program variables, column variables are read-only. You can use their existing value, but you cannot assign a new value to a column variable.

In the sample program, *&phone* is a column variable that you can use in SQR commands as if it were a string, date, or numeric variable, depending on its content. In the example condition, *&phone* is compared to '', which is an empty string. If *&phone* is an empty string, then the program prints *No phone*.

Changing Column Variable Names

Note that the *&phone* column variable illustrated in the previous section inherited its name from the phone column. This value is the default, but you can change it, as this example demonstrates:

```
begin-select
phone &cust_phone
   if &cust_phone = ''
      print 'No phone' (,1)
   else
```

```
      print &cust_phone (,1)
   end-if
   position (+1)
from customers
end-select
```

One reason for changing the name of the column variable is to use a selected column in an expression that has no name. For example:

```
begin-select
count(name) &cust_cnt (,1)
   if &cust_cnt < 100
      print 'Less than 100 customers'
   end-if
   position (+1)
from customers
group by city, state
end-select
```

In this example, the expression COUNT *(name)* is selected. In the program, you store this expression in the *&cust_cnt* column variable and refer to it afterwards by that name.

Chapter 6

Using Break Logic

Understanding Break Logic

A *break* is a change in the value of a column or variable. Records with the same value—for example, records with the same value for state—logically belong to a group. When a break occurs, a new group begins.

Use break logic in a report to:

- Add white space to reports.
- Avoid printing redundant data.
- Perform conditional processing on variables that change.
- Print subtotals.

For example, you can use break logic to prepare a sales report with records that are grouped by product, region, salesperson, or all three. Break logic also enables you to print column headings, count records, subtotal a column, and perform additional processing on a count or subtotal.

Here is a sample program without break logic:

```
Program ex5a.sqr
begin-program
   do list_customers
end-program
begin-heading 2
   print 'State' (1,1)
   print 'City'  (1,7)
   print 'Name'  (1,24)
   print 'Phone' (1,55)
end-heading
begin-procedure list_customers
begin-select
state (,1)
city  (,7)
name  (,24)
phone (,55)
   position (+1)  ! Advance to the next line
from customers
order by state, city, name
end-select
end-procedure ! list_customers
```

```
State City            Name                          Phone


IN    Davenport       Harold Alexander Fink         3015553645
IN    Miningville     Harry's Landmark Diner        3175550948
MI    Bell Harbor     Sam Johnson                   3135556732
NH    Frogline        Jerry's Junkyard Specialties  6125552877
NJ    Teaneck         Clair Butterfield             2015559901
NM    Big Falls       Joe Smith and Company         8085552124
NY    Mamaroneck      Harriet Bailey                9145550144
```

```
NY    New York         John Conway                     2125552311
NY    New York         Corks and Bottles, Inc.         2125550021
NY    New York         Kate's Out of Date Dress Shop   2125559000
NY    Queens           Eliot Richards                  2125554285
OH    Cleveland        Quentin Fields                  2165553341
OH    Everretsville    Gregory Stonehaven              2165553109
OH    Zanesville       Isaiah J Schwartz and Company   5185559813
```

When you sort output by state, city, and name (note the ORDER BY clause in the BEGIN-SELECT statement), the records are grouped by state. To make the grouping more apparent, you can add a break.

Using the ON-BREAK Option

In the following program, the ON-BREAK option of the PRINT command accomplishes two related tasks: it starts a new group each time the value of state changes, and it prints state only when its value changes. Note that ON-BREAK works as well for implicit as for explicit PRINT commands, such as in the following example, where state, city, name, and phone are implicitly printed as part of the select paragraph.

The sample program here is identical to ex5a.sqr except for the line that prints the state column, which appears like this:

```
Program ex5b.sqr
begin-program
   do list_customers
end-program
begin-heading 2
   print 'State' (1,1)
   print 'City'  (1,7)
   print 'Name'  (1,24)
   print 'Phone' (1,55)
end-heading
begin-procedure list_customers
begin-select
state (,1) on-break
city  (,7)
name  (,24)
phone (,55)
   position (+1)  ! Advance to the next line
from customers
order by state, city, name
end-select
end-procedure ! list_customers
```

The output for the ex5b.sqr program is:

```
State City             Name                            Phone


IN    Davenport        Harold Alexander Fink           3015553645
      Miningville      Harry's Landmark Diner          3175550948
MI    Bell Harbor      Sam Johnson                     3135556732
NH    Frogline         Jerry's Junkyard Specialties    6125552877
NJ    Teaneck          Clair Butterfield               2015559901
NM    Big Falls        Joe Smith and Company           8085552124
NY    Mamaroneck       Harriet Bailey                  9145550144
      New York         John Conway                     2125552311
      New York         Corks and Bottles, Inc.         2125550021
      New York         Kate's Out of Date Dress Shop   2125559000
      Queens           Eliot Richards                  2125554285
OH    Cleveland        Quentin Fields                  2165553341
      Everretsville    Gregory Stonehaven              2165553109
      Zanesville       Isaiah J Schwartz and Company   5185559813
```


With break processing, the state abbreviation is printed only once for each group.

Skipping Lines Between Groups

You can further enhance the visual effect of break processing by inserting one or more lines between groups. To do so, use the SKIPLINES qualifier with ON-BREAK. Here is the **list_customers** procedure from ex5b.sqr with the modified line shown **like this:**

```
begin-select
state (,1) on-break skiplines=1
city  (,7)
name  (,24)
phone (,55)
   position (+1)  ! Advance to the next line
from customers
order by state, city, name
end-select
```

The output for the modified ex5b.sqr program is:

```
State    City           Name                          Phone
IN       Davenport      Harold Alexander Fink         3015553645
         Miningville    Harry's Landmark Diner        3175550948


MI       Bell Harbor    Sam Johnson                   3135556732


NH       Frogline       Jerry's Junkyard Specialties  6125552877
.....
```

Arranging Multiple Break Columns

As you can see in the previous example, you can also have multiple customers within a city. You can apply the same break concept to the city column to make this grouping of customers more apparent. Add another ON-BREAK to the program so that city is also printed only when its value changes.

When you have multiple breaks, you must arrange them in a hierarchy. In the sample program, the breaks are for geographical units, so arranging them according to size is logical: first state and then city. This sort of arrangement is called *nesting,* and the breaks are considered nested.

To ensure that the breaks are properly nested, use the LEVEL keyword. This argument numbers breaks by level and specifies that the columns are printed in order of increasing break levels, from left to right. Number breaks in the same order in which they are sorted in the ORDER BY clause.

See Setting Break Procedures with BEFORE and AFTER Qualifiers.

The LEVEL argument enables you to control the order in which you call break procedures. The next sample program is identical to ex5a.sqr except for the two lines that print the state and city columns, which are shown in this way:

```
Program ex5c.sqr
begin-program
   do list_customers
end-program
begin-heading 2
```



```
   print 'State' (1,1)
   print 'City'  (1,7)
   print 'Name'  (1,24)
   print 'Phone' (1,55)
end-heading
begin-procedure list_customers
begin-select
state (,1) on-break level=1
city  (,7) on-break level=2
name  (,24)
phone (,55)
   position (+1)  ! Advance to the next line
from customers
order by state, city, name
end-select
end-procedure ! list_customers
```

The output for the ex5c.sqr program is:

```
State City            Name                          Phone

IN    Davenport       Harold Alexander Fink         3015553645
      Miningville     Harry's Landmark Diner        3175550948
MI    Bell Harbor     Sam Johnson                   3135556732
NH    Frogline        Jerry's Junkyard Specialties  6125552877
NJ    Teaneck         Clair Butterfield             2015559901
NM    Big Falls       Joe Smith and Company         8085552124
NY    Mamaroneck      Harriet Bailey                9145550144
      New York        John Conway                   2125552311
                      Corks and Bottles, Inc.       2125550021
                      Kate's Out of Date Dress Shop 2125559000
      Queens          Eliot Richards                2125554285
OH    Cleveland       Quentin Fields                2165553341
      Everretsville   Gregory Stonehaven            2165553109
      Zanesville      Isaiah J Schwartz and Company 5185559813
```

As you can see, three customers are in New York, so the city name for the second and third customers is left blank.

Using Break Processing Enhancements

This section discusses how to:

- Control page breaks and calculate subtotals and totals.
- Handle page breaks.
- Print the date.
- Obtain totals.
- Use hyphens and underscores.

Controlling Page Breaks and Calculating Subtotals and Totals

When you use break logic, you may want to enhance your report by controlling page breaks or calculating subtotals and totals for the ON-BREAK column. The following example illustrates these techniques.

The sample program selects the customer's name, address, and telephone number from the database. The break processing is performed on the state column:

```
Program ex5d.sqr
begin-program
  do list_customers
end-program
begin-heading 4
  print 'Customers Listed by State' (1) center
  print $current-date    (1,1) Edit 'DD-Mon-YYYY'
  print 'State' (3,1)
  print 'Customer Name, Address and Phone Number' (,11)
  print '-' (4,1,9) fill
  print '-' (4,11,40) fill
end-heading
begin-footing 2
  ! print "Page n of m"
  page-number (1,1) 'Page '
  last-page   () ' of '
end-footing
begin-procedure state_tot
  print '   Total Customers for State:  ' (+1,1)
  print #state_total () edit 999,999
  position (+3,1)     ! Leave 2 blank lines.
  let #cust_total = #cust_total + #state_total
  let #state_total = 0
end-procedure ! state_tot
begin-procedure list_customers
let #state_total = 0
let #cust_total = 0
begin-select
! The 'state' field will only be printed when it
! changes. The procedure 'state_tot' will also be
! executed only when the value of 'state' changes.
state       (,1)  on-break print=change/top-page after=state_tot
name        (,11)
addr1       (+1,11)   ! continue on second line
addr2       (+1,11)   ! continue on third line
city        (+1,11)   ! continue on fourth line
phone          (,+2) edit (xxx)bxxx-xxxx ! Edit for easy reading.
  ! Skip 1 line between listings.
  ! Since each listing takes 4 lines, we specify 'need=4' to
  ! prevent a customer's data from being broken across two pages.
  next-listing skiplines=1 need=4
  let #state_total = #state_total + 1
from customers
order by state, name
end-select
if #cust_total > 0
  print '   Total Customers: ' (+3,1)
  print #cust_total () edit 999,999  ! Total customers printed.
else
  print 'No customers.' (1,1)
end-if
end-procedure ! list_customers
```

The output for the ex5d.sqr program is:

```
29-Apr-2004

Customers Listed by State

State     Customer Name, Address and Phone Number
--------- ----------------------------------------
IN        Harold Alexander Fink
          32077 Cedar Street
          West End
          Davenport  (301) 555-3645
```



```
          Harry's Landmark Diner
          17043 Silverfish Road
          South Park
          Miningville  (317) 555-0948


Total Customers for State:        2


MI        Sam Johnson
          37 Cleaver Street
          Sandy Acres
          Bell Harbor  (313) 555-6732


 Total Customers for State:        1

NH        Jerry's Junkyard Specialties
          Crazy Lakes Cottages
          Rural Delivery #27
          Frogline  (612) 555-2877


Total Customers for State:        1

...
```

Take a close look at the code. The data is printed by using a select paragraph in the **list_customer** procedure. The state and the customer name are printed on the first line. The customer's address and phone number are printed on the next three lines.

The program also uses the argument AFTER=STATE_TOT. This argument calls the **state_tot** procedure after each change in the value of state.

See Setting Break Procedures with BEFORE and AFTER Qualifiers

Handling Page Breaks

If a page break occurs within a group, you may want to reprint headings and the value of the break column at the top of the new page.

To control the printing of the value, use PRINT=CHANGE/TOP-PAGE. With this qualifier, the value of the ON-BREAK column is printed when it changes and after every page break. In this example, the value of state is printed not only when it changes, but whenever the report starts a new page.

To format records, use the NEXT-LISTING command. This command serves two purposes: the SKIPLINES=1 argument skips one line between records and then renumbers the current line as line 1; the NEED=4 argument prevents a listing from being split over two pages by specifying the minimum number of lines that are needed to write a new listing on the current page. In this case, if fewer than four lines are left on a page, SQR starts a new page.

Printing the Date

In the HEADING section, the reserved variable *$current-date* prints the date and the time. This variable is initialized with the date and time of the client machine when the program starts to run. SQR provides predefined, or reserved, variables for a variety of uses.

In this example, the complete command is `PRINT $current-date (1,1) EDIT 'DD/Mon/YYYY'`. It prints the date and time at position 1,1 of the heading. The EDIT argument specifies an edit mask, or format, for printing the date. SQR provides a variety of edit masks for use in formatting numbers, dates, and strings.

See "PRINT" (PeopleTools 8.58: SQR Language Reference for PeopleSoft).

Note that the PRINT command for the report title precedes the command for the *$current-date* reserved variable, even though the date is on the left and the title is on the right. SQR always assembles a page in memory before printing, so the order of these commands does not matter if you use the correct print position qualifiers.

The last two commands in the HEADING section print a string of hyphens under the column headings. Note the use of the FILL option with the PRINT command. This option tells SQR to fill the specified width with this pattern, which is a useful method to print a line.

The FOOTING section prints *Page n of m* as in earlier examples.

Related Links

SQR for PeopleSoft Developers

Obtaining Totals

The ex5d.sqr program also prints a subtotal of customers in each state and a grand total of all customers. These calculations are performed with two numeric variables, one for the subtotal and one for the grand total. These variables are:

- *#state_total*
- *#cust_total*

SQR for PeopleSoft has a small set of variable types. The most common types are numeric variables and string variables. All numeric variables in SQR are preceded by a pound sign (#), and all string variables are preceded by a dollar sign ($). An additional SQR variable type is the date variable.

In SQR for PeopleSoft, numeric and string variables are not explicitly declared. Instead, they are implicitly defined by their first use. All numeric variables start out as zero and all string variables start out as null, so they do not need to be initialized. The string variables are of varying length and can hold long and short strings of characters. Assigning a new value to a string variable automatically adjusts its length.

In the **list_customers** procedure, *#state_total* and *#cust_total* are set to zero at the beginning of the procedure. This initialization is optional and is done for clarity only. The *#state_total* variable increments by 1 for every row that is selected.

When the value of state changes, the program calls the **state_tot** procedure and prints the value of *#state_total*. Note the use of the `EDIT 999,999` edit mask, which formats the number.

This procedure also employs the LET command. LET is the assignment command in SQR for building complex expressions. Here, LET adds the value of *#state_total* to *#cust_total*. At the end of the procedure, *#state_total* is reset to zero.

The **list_customers** procedure contains an example of the SQR if-then-else logic. The condition starts with IF followed by an expression. If the expression evaluates to true or to a number other than zero,

the subsequent commands are run. Otherwise, if the IF command has an ELSE command, then those commands are run. IF commands always end with an END-IF command.

In ex5d.sqr, the value of *#cust_total* is examined. If it is greater than zero, the query has returned rows of data, and the program prints the string *Total Customers:* and the value of *#cust_total*.

If *#cust_total* is zero, the query has not returned any data. In that case, the program prints the string *No customers.*

Using Hyphens and Underscores

Many SQR commands, such as BEGIN-PROGRAM and BEGIN-SELECT, use a hyphen, whereas procedure and variable names use an underscore.

Procedure and variable names can contain either a hyphen or underscore, but you should use underscores in procedure and variable names to distinguish them from SQR commands. Doing so also prevents confusion when you mix variable names and numbers in an expression, where hyphens could be mistaken for minus signs.

Setting Break Procedures with BEFORE and AFTER Qualifiers

When you print variables with ON-BREAK, you can automatically call procedures before and after each break in a column. The BEFORE and AFTER qualifiers provide this capability. For example:

```
begin-select
state (,1) on-break before=state_heading after=state_tot
```

The BEFORE qualifier automatically calls the **state_heading** procedure to print headings before each group of records of the same state. Similarly, the AFTER qualifier automatically calls the **state_tot** procedure to print totals after each group of records.

All BEFORE procedures are automatically invoked before each break, including the first; that is, before the select paragraph is even processed. Similarly, all AFTER procedures are invoked after each break, including the last group; that is, upon completion of the select paragraph.

Order of Events

You can define a hierarchy of break columns by using the LEVEL qualifier of ON-BREAK. In the ex5c.sqr sample program, for example, state was defined as LEVEL=1 and city as LEVEL=2.

When a break occurs at one level, it also forces breaks on variables with higher LEVEL qualifiers. In the sample program, a break on state also means a break on city.

A break on a variable can initiate many other events. The value can be printed, lines can be skipped, procedures can be called automatically, and the old value can be saved. Knowing the order of events is important, particularly when multiple ON-BREAK columns exist.

The following select paragraph has breaks on three levels:

```
begin-select
state  (,1)   on-break level=1      after=state_tot   skiplines=2
city (,7)     on-break level=2      after=city_tot  skiplines=1
zip   (,45)   on-break level=3      after=zip_tot
from customers
```



```
order by state, city, zip
end-select
```

The system processes breaks in the following way:

1. When zip breaks, the **city_tot** procedure is run.
2. When city breaks, first the **zip_tot** procedure is run, and then the **city_tot** procedure is run and one line is skipped (SKIPLINES=1).

   Both city and zip are printed in the next record.
3. When state breaks, the zip_tot, city_tot, and state_tot procedures are processed in that order.

   One line is skipped after the **city_tot** procedure is run, and two lines are skipped after the **state_tot** procedure is run. All three columns—state, city, and zip—are printed in the next record.

The following program (ex5e.sqr) demonstrates the order of events in break processing. It has three ON-BREAK columns, each with a LEVEL argument and a BEFORE and AFTER procedure. The BEFORE and AFTER procedures print strings to indicate the order of processing.

```
Program ex5e.sqr
begin-setup
      declare-Layout
       default
      end-declare
end-setup
begin-program
 do main
end-program
begin-procedure a
print 'AFTER Procedure for state LEVEL 1' (+1,40)
end-procedure
begin-procedure b
print 'AFTER Procedure city LEVEL 2' (+1,40)
end-procedure
begin-procedure c
print 'AFTER Procedure zip LEVEL 3' (+1,40)
end-procedure
begin-procedure aa
print 'BEFORE Procedure state LEVEL 1' (+1,40)
end-procedure
begin-procedure bb
print 'BEFORE Procedure city LEVEL 2' (+1,40)
end-procedure
begin-procedure cc
print 'BEFORE Procedure zip LEVEL 3' (+1,40)
end-procedure
begin-procedure main local
begin-select
      add 1 to #count
      print 'Retrieved row #' (+1,40)
      print #count (,+10)Edit 9999
      position (+1)
state   (3,1) On-Break Level=1 after=a before=aa
city    (3,10) On-Break Level=2 after=b before=bb
zip     (3,25) On-Break Level=3 after=c before=cc Edit xxxxx
 next-listing  Need=10
from customers
order by state,city,zip
end-select
end-procedure
begin-heading 3
 print $current-date (1,1) edit 'DD-MM-YYYY'
 page-number (1,60) 'Page '
 last-page () ' of '
 print 'STATE'  (3,1)
```



```
 print 'CITY'   (3,10)
 print 'ZIP'    (3,25)
 print 'Break Processing sequence' (3,40)
end-heading
```

The output for the ex5e.sqr program is:

```
02-05-2004                                          Page 1 of 3


STATE    CITY           ZIP        Break Processing sequence

              BEFORE Procedure state LEVEL 1
IN       Davenport      62130      BEFORE Procedure city LEVEL 2
                                   BEFORE Procedure zip LEVEL 3

                                   Retrieved row #1
                Retrieved row #2
         Miningville    40622
                                   AFTER Procedure zip LEVEL 3
                                   AFTER Procedure city LEVEL 2
                                   BEFORE Procedure city LEVEL 2
                                   BEFORE Procedure zip LEVEL 3

                Retrieved row #3
MI       Bell Harbor    40674
                                   AFTER Procedure zip LEVEL 3
                                   AFTER Procedure city LEVEL 2
                                   AFTER Procedure for state LEVEL 1
                                   BEFORE Procedure state LEVEL 1
                                   BEFORE Procedure city LEVEL 2
                                   BEFORE Procedure zip LEVEL 3

                Retrieved row #4
NH       Frogline       04821
                                   AFTER Procedure zip LEVEL 3
                                   AFTER Procedure city LEVEL 2
                                   AFTER Procedure for state LEVEL 1
                                   BEFORE Procedure state LEVEL 1
                                   BEFORE Procedure city LEVEL 2
                                   BEFORE Procedure zip LEVEL 3

                                   Retrieved row #5
NJ       Teaneck        00355
                                   AFTER Procedure zip LEVEL 3
                                   AFTER Procedure city LEVEL 2
                                   AFTER Procedure for state LEVEL 1
                                   BEFORE Procedure state LEVEL 1
                                   BEFORE Procedure city LEVEL 2
                                   BEFORE Procedure zip LEVEL 3

                                   Retrieved row #6
NM       Big Falls      87893
                                   AFTER Procedure zip LEVEL 3
                                   AFTER Procedure city LEVEL 2
                                   AFTER Procedure for state LEVEL 1
                                   BEFORE Procedure state LEVEL 1
                                   BEFORE Procedure city LEVEL 2
                                   BEFORE Procedure zip LEVEL 3

02-05-2004                                          Page 2 of 3

STATE    CITY           ZIP        Break Processing sequence

                Retrieved row #7
NY       Mamaroneck     10833
                                   AFTER Procedure zip LEVEL 3
                                   AFTER Procedure city LEVEL 2
                                   AFTER Procedure for state LEVEL 1
                                   BEFORE Procedure state LEVEL 1
                                   BEFORE Procedure city LEVEL 2
```



```
                                BEFORE Procedure zip LEVEL 3
```

. . .

The following steps explain the order of processing in detail:

1. Process BEFORE procedures.

   BEFORE procedures are processed in ascending order by LEVEL before the first row of the query is retrieved. If no data is selected, BEFORE procedures are not run.

2. Select the first row of data.

3. Select subsequent rows of data.

   Processing of the select paragraph continues. When a break occurs on any column, it also initiates breaks on columns at the same or higher levels. Events occur in the following order:

   a. AFTER procedures are processed in descending order from the highest level to the level of the current ON-BREAK column.

   b. SAVE variables are set with the value of the previous ON-BREAK column.

   c. BEFORE procedures are processed in ascending order from the current level to the highest level.

   d. If SKIPLINES was specified, the current line position is advanced.

   e. The value of the new group is printed (unless PRINT=NEVER is specified).

4. Process AFTER procedures.

   After the select paragraph is complete, if any rows were selected, AFTER procedures are processed in descending order by LEVEL.

See Saving a Value When a Break Occurs.

Controlling Page Breaks with Multiple ON-BREAK Columns

When multiple columns have ON-BREAK, page breaks need careful planning. While having a page break within a group, you probably would not want to have one within a record.

You can prevent page breaks within a record by following four simple rules:

- Place ON-BREAK columns ahead of other columns in the select paragraph.
- Place the lower-level ON-BREAK columns ahead of the higher-level ON-BREAK columns in the select paragraph.
- Use the same line positions for all ON-BREAK columns.
- Avoid using WRAP and ON-BREAK together on one column.

Saving a Value When a Break Occurs

In ex5d.sqr, the **state_tot** procedure prints the total number of customers per state. Because it is called with the AFTER argument, this procedure is run only after the value of the ON-BREAK column, state, has changed.

Sometimes, however, you may want to print the previous value of the ON-BREAK column in the AFTER procedure. For example, you may want to print the state name and the totals for each state. Printing the value of state will not work because its value will have changed by the time the AFTER procedure is called.

The solution is to save the previous break value in a string variable. To do this, use the SAVE qualifier of ON-BREAK. For example:

```
begin-select
state (,1) on-break after=state_tot save=$old_state
```

You can then print the value of *$old_state* in the **state_tot** procedure.

Using ON-BREAK on a Hidden Column

In some reports, you may want to use the features of break processing without printing the ON-BREAK option. For example, you may want to incorporate the ON-BREAK option into a subheading. This format might make your report more readable. It is also useful when you want to leave room on the page for additional columns.

To create such a report, you can hide the break option using the PRINT=NEVER qualifier and print it in a heading procedure that is called by BEFORE.

The following code is based on the ex5b.sqr program, with the key lines shown like this:

```
Program ex5f.sqr

begin-program
   do list_customers
end-program
begin-procedure list_customers
begin-select
state () on-break before=state_heading print=never level=1
city  (,1) on-break level=2
name  (,18)
phone (,49)
   position (+1) ! Advance to the next line
from customers
order by state, city, name
end-select
end-procedure ! list_customers
begin-procedure state_heading
   print 'State: ' (+1,1) bold    ! Advance a line and print 'State:'
   print &state (,8) bold         ! Print the state column here
   print 'City' (+1,1) bold       ! Advance a line and print 'City'
   print 'Name' (,18) bold
   print 'Phone' (,49) bold
   print '-' (+1,1,58) fill
   position (+1)                  ! Advance to the next line
end-procedure ! state_heading
```

Note: This program has no HEADING section. Instead, a procedure prints column headings for each state rather than at the top of each page. The *&state* variable can be referenced throughout the program, even though the state column was not printed as part of the break.

Examine the following line in the program from the select paragraph:

```
state () on-break before=state_heading print=never level=1
```

This line defines the break processing for state. The BEFORE qualifier specifies that the **state_heading** procedure is automatically called when the state changes. In this program, the break is set to LEVEL=1.

The PRINT=NEVER qualifier hides the state column and specifies that it is not printed as part of the select paragraph. Instead, it is printed in the **state_heading** procedure. In this procedure, the state column is referred to as the *&state* column variable.

The city column is assigned a LEVEL=2 break.

The output for the ex5f.sqr program is:

```
State: IN
City             Name                              Phone
---------------------------------------------------------
Davenport        Harold Alexander Fink             3015553645
Miningville      Harry's Landmark Diner            3175550948


State: MI
City             Name                              Phone
---------------------------------------------------------
Bell Harbor      Sam Johnson                       3135556732


State: NH
City             Name                              Phone
---------------------------------------------------------
Frogline         Jerry's Junkyard Specialties      6125552877


State: NJ
City             Name                              Phone
---------------------------------------------------------
Teaneck          Clair Butterfield                 2015559901


State: NM
City             Name                              Phone
---------------------------------------------------------
Big Falls        Joe Smith and Company             8085552124


State: NY
City             Name                              Phone
---------------------------------------------------------
Mamaroneck       Harriet Bailey                    9145550144
New York         John Conway                       2125552311
                 Corks and Bottles, Inc.           2125550021
                 Kate's Out of Date Dress Shop     2125559000
Queens           Eliot Richards                    2125554285


State: OH
City             Name                              Phone
---------------------------------------------------------
Cleveland        Quentin Fields                    2165553341
Everretsville    Gregory Stonehaven                2165553109
Zanesville       Isaiah J Schwartz and Company     5185559813
```


Performing Break Processing on Numeric Values

You cannot use ON-BREAK with SQR numeric variables. To perform break processing on a numeric variable, you must first move its value to a string variable and then set ON-BREAK on that. For example:

```
begin-select
amount_received &amount
  move &amount to $amount $$9,999.99
  print $amount (+1,1) on-break
from cash_receipts
order by amount_received
end-select
```

The maximum number of ON-BREAK levels is determined by the ON-BREAK setting in the [Processing-Limits] section of the PSSQR.INI file. The default is 30, but you can increase this setting. Its maximum value is 64K-1 (65,535).

Related Links

SQR Language Reference for PeopleSoft

Chapter 7

Adding Declarations Using the SETUP Section

Understanding the SETUP Section

You place all declarations in a SETUP section. Declarations define certain report characteristics and the source and attributes of various report components, such as charts and images. The SETUP section is evaluated when you compile the program, before you run the program. A program is not required to have a SETUP section, but it can be useful.

Creating a SETUP Section

Place a SETUP section at the beginning of the program, before the PROGRAM section. Begin the section with a BEGIN-SETUP paragraph and end it with an END-SETUP paragraph.

Use the following commands in the SETUP section:

Command	***Comment***
ALTER-LOCALE	Can also appear in a procedure.
ASK	Allowed only in a SETUP section.
BEGIN-SQL	Can also appear in a procedure. Processed when a runtime file (with .SQT extension) is loaded.
CREATE-ARRAY	Can also appear in a procedure.
DECLARE-CHART	NA
DECLARE-IMAGE	NA
DECLARE-LAYOUT	NA
DECLARE-PRINTER	NA
DECLARE-PROCEDURE	NA
DECLARE-REPORT	NA
DECLARE-TOC	NA
DECLARE-VARIABLE	Can also appear in a local procedure.

Command	*Comment*
LOAD-LOOKUP	Can also appear in a procedure.

Related Links

SQR Language Reference for PeopleSoft

Using the DECLARE-LAYOUT Command

Use the DECLARE-LAYOUT command to set the page layout and to include important options, such as the paper size and margins.

Sample SETUP Program

Here is a typical SETUP section:

```
begin-setup
   ! Declare the default layout for this report
   declare-layout default
      paper-size=(8.5,11)
      left-margin=1    right-margin=1
      top-margin=1     bottom-margin=1
   end-declare
end-setup
```

In the preceding example, the DECLARE-LAYOUT command sets the paper size to 8 1/2 by 11 inches, with all margins at 1 inch.

In SQR for PeopleSoft, data is positioned on the page using line and character position coordinates. Think of the page as a grid and each cell in the grid holds one character. With such a grid, in a position qualifier consisting of *(line, column, width), column* and *width* are numbers that denote characters and spaces.

Defining the SQR Page Layout

The main attributes of the DECLARE-LAYOUT command affect the structure of a page.

The PAPER-SIZE argument defines the dimensions of the entire page, including the margins. The TOP-MARGIN, LEFT-MARGIN, BOTTOM-MARGIN, and RIGHT-MARGIN arguments define the margins. In SQR, you cannot print in the margins.

In the preceding sample program, the left margin uses 10 spaces and the top margin uses 6 lines. The page width accommodates 65 characters (without the margins) and 54 lines.

The default mapping of characters and lines to inches is 10 characters per inch and six lines per inch . These dimensions mean that each character cell is 1/10 inch wide and 1/6 inch high. These settings are used when a program does not contain a DECLARE-LAYOUT command.

Overriding Default Settings

Override the default settings by using the LINE-HEIGHT and CHAR-WIDTH arguments in the DECLARE-LAYOUT command. These arguments adjust the dimensions of a grid, implying a change in the meaning of column and line. If the DECLARE-LAYOUT paragraph includes the LINE-HEIGHT=1 and CHAR-WIDTH=1 arguments, then the cells in the grid measure 1 point by 1 point (1 point is 1/72 inch or approximately 0.35 millimeters). In that case, column is a dimension described in points. The length of a string, however, is still described in characters.

Alternatively, you can use the MAX-LINES and MAX-COLUMNS arguments of the DECLARE-LAYOUT command to specify the number of lines on a page and the number of characters that will fit across the page. SQR calculates the line height and character width based on these settings and the size of the page and margins.

Specify coordinates in terms of lines and character positions. The first line from the top is 1, and the first column (from the left) is 1. No coordinate 0 exists.

Declaring a Page Orientation

Use the DECLARE-LAYOUT command to declare a page orientation. Note that this declaration does not affect how SQR uses position coordinates. Line and character positions are not transposed when page orientation is switched. The only effect of the ORIENTATION option of the DECLARE-LAYOUT command is that SQR switches the printer to the specified orientation: portrait or landscape. The default mode is portrait.

Chapter 8

Creating Master and Detail Reports

Understanding Master and Detail Reports

Master and detail reports show hierarchical information. The information is normally retrieved from multiple tables that have a one-to-many relationship, such as customers and orders. The customer information is the master, and the orders are the detail.

Often, you can obtain such information with a single SQR select paragraph. In such a program, the data from the master table is joined with data from the detail table. You can implement break logic to group the detail records for each master record. This type of report has one major disadvantage: if a master record has no associated detail records, then the system does not display it. If you need to show all master records, whether they have detail records or not, this type of report will not meet your needs.

See Understanding Break Logic.

To show all master records, whether or not they have detail records, create a master and detail report with one SELECT statement that retrieves records from the master table, followed by separate SELECT statements that retrieve the detail records that are associated with each master record.

The sample program for Master and Detail Reports produces just such a report. In the example, one BEGIN-SELECT command returns the names of customers. For each customer, two additional BEGIN-SELECT commands are run—one to retrieve order information and another to retrieve payment information.

When one query returns master information and another query returns detail information, the detail query is nested within the master query.

Understanding the Sample Program for Master and Detail Reports

In the sample program, nested queries are invoked once for each customer, and each one retrieves records that correspond to the current customer. A bind variable correlates the subqueries in the WHERE clause. This variable correlates the customer number (cust_num) with the current customer record:

```
Program ex7a.sqr
begin-program
  do main
end-program
begin-procedure main
begin-select
  Print 'Customer Information' (,1)
  Print '-'                    (+1,1,45) Fill
name   (+1,1,25)
city   (,+1,16)
state  (,+1,2)
cust_num
  do cash_receipts(&cust_num)
  do orders(&cust_num)
  position (+2,1)
```

```
from customers
end-select
end-procedure ! main
begin-procedure cash_receipts (#cust_num)
  let #any = 0
begin-select
  if not #any
     print 'Cash Received' (+2,10)
     print '-------------' (+1,10)
     let #any = 1
  end-if
date_received    (+1,10,20) edit 'DD-MON-YY'
amount_received  (,+1,13) Edit $$$$,$$0.99
from cash_receipts a
where a.cust_num = #cust_num
end-select
end-procedure ! cash_receipts
begin-procedure orders (#cust_num)
  let #any = 0
begin-select
  if not #any
    print 'Orders Booked' (+2,10)
    print '-------------' (+1,10)
    let #any = 1
  end-if
a.order_num
order_date                (+1,10,20) Edit 'DD-MON-YY'
description               (,+1,20)
c.price * b.quantity      (,+1,13) Edit $$$$,$$0.99
from  orders a, ordlines b, products c
where a.order_num = b.order_num
  and b.product_code = c.product_code
  and a.cust_num = #cust_num
end-select
end-procedure ! orders
begin-heading 3
 print $current-date (1,1) Edit 'DD-MON-YYYY'
 page-number (1,69) 'Page '
end-heading
```

Correlating Subqueries

The ex7a.sqr sample program contains three procedures—main, cash_receipts, and orders—that correspond to the three queries. The main procedure is the master. It retrieves the customer names. For each customer, the program invokes the cash_receipts procedure to list the cash receipts, if any, and the orders procedure to list the customer's orders, if any.

The procedures take the *cust_num* variable as an argument. As you can see, cash_receipts and orders are called many times, once for each customer. Each time, the procedures perform the same query with a different value for the *cust_num* variable in the WHERE clause.

Note the use of the IF command and the *#any* numeric variable in these procedures. When the BEGIN-SELECT command returns no records, SQR does not process the PRINT commands that follow. Thus, the headings for these procedures appear only for those customers who have records in the detail tables.

The orders procedure demonstrates the use of an expression in the BEGIN-SELECT command. The expression is `c.price * b.quantity.`

Note: Examine the format of the dollar amount with the argument `EDIT $$$$,$$0.99.` This format uses a "floating-to-the-right" money symbol. If fewer digits are used than the six that we specified here, the dollar sign floats to the right and remains close to the number.

See Using Variables in SQL.

Sample Program Output

The following is the output for program ex7a.sqr:

```
6-APR-2004                                                    Page 1

Customer Information
-------------------------------------------
Gregory Stonehaven        Everretsville    OH

         Cash Received
         -------------
         01-FEB-03               $130.00

Customer Information
-------------------------------------------
John Conway               New York         NY

         Cash Received
         -------------
         01-MAR-03               $140.00

Customer Information
-------------------------------------------
Eliot Richards            Queens           NY

         Cash Received
         -------------
         16-JAN-03               $220.12
         17-JAN-03               $260.00

         Orders Booked
         -------------
         02-MAY-03           Whirlybobs              $239.19
         02-MAY-03           Canisters             $3,980.25

Customer Information
-------------------------------------------
Isaiah J Schwartz and Com Zanesville       OH

         Cash Received
         -------------
         18-JAN-03               $190.00
         02-JAN-03             $1,100.00

         Orders Booked
         -------------
         02-MAY-03           Hop scotch kits         $6,902.00
         02-MAY-03           Wire rings             $19,872.90

Customer Information
-------------------------------------------
Harold Alexander Fink     Davenport        IN

         Cash Received
         -------------
         01-FEB-03             $1,200.00
         01-MAR-03             $1,300.00

         Orders Booked
         -------------
         19-MAY-03           Ginger snaps             $44.28
         19-MAY-03           Modeling clay           $517.05
```


Chapter 9

Creating Cross-Tabular Reports

Understanding Cross-Tabular Reports

Cross-tabular reports are matrix-like or spreadsheet-like reports. These reports are useful for presenting summary numeric data. Cross-tabular reports vary in format. The following example shows sales revenue summarized by product by sales channel:

```
Revenue by product by sales channel

Product    Direct Sales Resellers Mail Order    Total
---------- ------------ --------- ----------- -------
A                 2,100     1,209           0   3,309
B                   120       311         519     950
C                     2         0         924     926
---------- ------------ --------- ----------- -------
Total             2,222     1,520       1,443   5,185
```

This report is based on many sales records. The three middle columns correspond to sales channel categories. Each row corresponds to a product. The records fall into nine groups: three products sold through three sales channels. Some groups have no sales (such as mail order for product A).

Each category can be a discrete value of some database column or a set of values. For example, Resellers can be domestic resellers plus international distributors.

A category can also represent a range, as shown in this example:

```
              Orders by Product by Order Size
Product
Category    Less than 10 10 to 100 More than 100   Total
----------- ------------ --------- ------------- -------
Durable              200       120             0     320
Nondurable           122       311           924    1876
----------- ------------ --------- ------------- -------
Total                322       431          1443    2196
```

In this example, the rows correspond to the categories Durable and Nondurable. The columns represent ranges of order size.

For each record that is selected, the program must determine the range to which it belongs and add 1 to the count for that category. The numbers in the cells are counts, but they could be sums, averages, or any other expression.

Of course, other types of cross-tabular reports exist. These reports become more complex when the number of columns is not predefined and when more columns exist than can fit across a page.

Using an Array

Often, the program must process all of the records before it can begin to print the data. During processing, the program must keep the data in a buffer where it can accumulate the numbers. This can be done in an SQR array.

An *array* is a unit of storage that contains rows and columns. An array is similar to a database table, but it exists only in memory.

The sample program specifies an array called order_qty to hold the sum of the quantity of orders in a given month. You could program this specific example without an array, but using one can be beneficial. Data that you retrieve once and store in an array can be presented in many ways without additional database queries. The data can even be presented in a chart.

The sample program also demonstrates an SQR feature called a three-dimensional array. This type of array has fields (columns) and rows, and it also has repeating fields (the third dimension). In the order_qty array, the first field is the product description. The second field is the order quantity of each month. The example includes three months; therefore, this field repeats three times.

SQR references arrays in expressions such as `array_name.field(sub1[,sub2])`. The first subscript, `sub1,` is the row number. The row count starts with zero. The second subscript, `sub2,` is specified when the field repeats. Repeating fields are also numbered starting with zero. The subscript can be a literal or an SQR numeric variable.

```
program ex8a.sqr

#define max_products 100
begin-setup
  create-array
     name=order_qty       size={max_products}
     field=product:char   field=month_qty:number:3
end-setup
begin-program
  do select_data
  do print_array
end-program
begin-procedure print_array
  let #entry_cnt = #i
  let #i = 0
  while #i <= #entry_cnt
      let $product  = order_qty.product(#i)
      let #jan      = order_qty.month_qty(#i,0)
      let #feb      = order_qty.month_qty(#i,1)
      let #mar      = order_qty.month_qty(#i,2)
      let #prod_tot = #jan + #feb + #mar
      print $product  (,1,30)
      print #jan      (,32,9) edit 9,999,999
      print #feb      (,42,9) edit 9,999,999
      print #mar      (,52,9) edit 9,999,999
      print #prod_tot (,62,9) edit 9,999,999
      position (+1)
      let #jan_total = #jan_total + #jan
      let #feb_total = #feb_total + #feb
      let #mar_total = #mar_total + #mar
      let #i = #i + 1
  end-while
  let #grand_total = #jan_total + #feb_total + #mar_total
  print 'Totals'     (+2,1)
  print #jan_total   (,32,9) edit 9,999,999
  print #feb_total   (,42,9) edit 9,999,999
  print #mar_total   (,52,9) edit 9,999,999
  print #grand_total (,62,9) edit 9,999,999
```



```
end-procedure print_array
begin-procedure select_data
begin-select
order_date
! The quantity for this order
quantity
! the product for this order
description
  if #i = 0 and order_qty.product(#i) = ''
      let order_qty.product(#i) = &description
  end-if
  if order_qty.product(#i) != &description
      let #i = #i + 1
      if #i >= {max_products}
         display 'Error: There are more than {max_products} products'
         stop
      end-if
      let order_qty.product(#i) = &description
  end-if
  let #j = to_number(datetostr(&order_date,'MM')) - 1
  if #j < 3
      let order_qty.month_qty(#i,#j) =
               order_qty.month_qty(#i,#j) + &quantity
  end-if
from  orders a, ordlines b, products c
where a.order_num = b.order_num
and   b.product_code = c.product_code
order by description
end-select
end-procedure ! select_data
begin-heading 4
  print $current-date (1,1)
  print 'Order Quantity by Product by Month' (1,18)
  page-number (1,64) 'Page '
  print 'Product'   (3,1)
  print '  January' (,32)
  print ' February' (,42)
  print '    March' (,52)
  print '    Total' (,62)
  print '-'         (4,1,70) Fill
end-heading
```

The following output is for program ex8a.sqr:

```
11-JUN-04      Order Quantity by Product by Month           Page 1

Product                      January   February     March     Total
-------------------------------------------------------------------
Canisters                          3          0         0         3
Curtain rods                       2          8        18        28
Ginger snaps                       1         10         0        11
Hanging plants                     1         20         0        21
Hookup wire                       16         15         0        31
Hop scotch kits                    2          0         0         2
Modeling clay                      5          0         0         5
New car                            1          9         0        10
Thimble                            7         20         0        27
Thingamajigs                      17          0       120       137
Widgets                            4          0        12        16
Wire rings                         1          0         0         1
Totals                            60         82       150       292
```

See Understanding Business Charts.

Creating an Array

You must define the size of an array when you create it. The sample program creates the order_qty array with a size of 100.

The #DEFINE MAX_PRODUCTS 100 command defines the max_products constant as a substitution variable. The sample program uses this constant to define the size of the array. Using #DEFINE is a good practice because it displays the limit at the top of the program source. Otherwise, it would be hidden in the code.

The SETUP section creates the array by using the CREATE-ARRAYcommand. All SQR arrays are created before the program begins running. Their size must be known at compile time. If you do not know exactly how many rows you have, you must over-allocate and specify an upper bound. In the example, the array has 100 rows even though the program uses only 12 rows to process the sample data.

The preceding program has two procedures: **select_data** and **print_array. Select_data** performs the database query, as its name suggests. While the database records are being processed, no data prints and the data accumulates in the array. When the processing is complete, the **print_array** procedure does two things: the procedure loops through the array and prints the data, and it also adds the month totals and prints them at the bottom.

The report summarizes the product order quantities for each month, which are the records ordered by the product description. The procedure then fills the array one product at a time. For each record that is selected, the procedure checks to see whether it is a new product; if it is, the array is incremented by row subscript `#i.` The procedure also adds the quantity to the corresponding entry in the array based on the month.

This program has one complication: how to obtain the month. Date manipulation can vary among databases, and to write truly portable code requires careful planning.

The key is the datetostr function in the following command:

```
let #j = to_number(datetostr(&order_date, 'MM')) - 1
```

This function converts the order_date column into a string. (The ‘MM’ edit mask specifies that only the month part be converted.) The resulting string is then converted to a number; if it is less than 3, it represents January, February, or March and is added to the array.

Grouping by Category

The following output is a cross-tabular report that groups the products by price range. This grouping cannot be done by using a SQL GROUP BY clause. Moreover, to process the records in order of price category, the program would have to sort the table by price. The sample program shows how to do it without sorting the data.

The sample program uses an SQR EVALUATE command to determine the price category and assign the array subscript `#i` to 0, 1, or 2. Then it adds the order quantity to the array cell that corresponds to the price category (row) and the month (column).

```
Program ex8b.sqr

#define max_categories 3
```



```
begin-setup
  create-array
     name=order_qty       size={max_categories}
     field=category:char  field=month_qty:number:3
end-setup
begin-program
  do select_data
  do print_array
end-program
begin-procedure print_array
  let #i = 0
  while #i < {max_categories}
      let $category = order_qty.category(#i)
      let #jan      = order_qty.month_qty(#i,0)
      let #feb      = order_qty.month_qty(#i,1)
      let #mar      = order_qty.month_qty(#i,2)
      let #category_tot = #jan + #feb + #mar
      print $category     (,1,31)
      print #jan          (,32,9) edit 9,999,999
      print #feb          (,42,9) edit 9,999,999
      print #mar          (,52,9) edit 9,999,999
      print #category_tot (,62,9) edit 9,999,999
      position (+1)
      let #jan_total = #jan_total + #jan
      let #feb_total = #feb_total + #feb
      let #mar_total = #mar_total + #mar
      let #i = #i + 1
  end-while
  let #grand_total = #jan_total + #feb_total + #mar_total
  print 'Totals'     (+2,1)
  print #jan_total   (,32,9) edit 9,999,999
  print #feb_total   (,42,9) edit 9,999,999
  print #mar_total   (,52,9) edit 9,999,999
  print #grand_total (,62,9) edit 9,999,999
end-procedure print_array
begin-procedure select_data
  let order_qty.category(0) = '$0-$4.99'
  let order_qty.category(1) = '$5.00-$100.00'
  let order_qty.category(2) = 'Over $100'
begin-select
order_date
! the price / price category for the order
c.price &price
  move &price to #price_num
  evaluate #price_num
  when < 5.0
     let #i = 0
     break
  when <= 100.0
     let #i = 1
     break
  when-other
     let #i = 2
     break
  end-evaluate
! The quantity for this order
quantity
  let #j = to_number(datetostr(&order_date,'MM')) - 1
  if #j < 3
      let order_qty.month_qty(#i,#j) =
                order_qty.month_qty(#i,#j) + &quantity
  end-if
from  orders a, ordlines b, products c
where a.order_num = b.order_num
and   b.product_code = c.product_code
end-select
end-procedure ! select_databegin-heading 5
  print $current-date (1,1)
  page-number (1,64) 'Page '
  print 'Order Quantity by Product Price Category by Month' (2,11)
  print 'Product Price Category'   (4,1)
```



```
  print '  January' (,32)
  print ' February' (,42)
  print '    March' (,52)
  print '    Total' (,62)
  print '-'         (5,1,70) Fill
end-heading
```

The following is the output for program ex8b.sqr:

```
11-JUN-04                                                    Page 1
          Order Quantity by Product Price Category by Month

Product Price Category        January  February   March     Total
-------------------------------------------------------------------
0-4.99                          28       45       12        85
5.00-100.00                     25       28      138       191
Over 100                         7        9        0        16

Totals                          60        82      150      292
```

Using Multiple Arrays

Using SQR arrays to buffer data offers several advantages. In the previous example, it eliminated the need to sort the data. Another advantage is that you can combine the two sample reports into one. With one pass on the data, you can fill the two arrays and then print the two parts of the report.

The following sample program performs the work that is done by the first two programs. The SETUP section specifies two arrays: one to summarize monthly orders by product and another to summarize monthly orders by price range.

```
Program ex8c.sqr

#define max_categories 3
#define max_products 100
begin-setup
  create-array
     name=order_qty       size={max_products}
     field=product:char   field=month_qty:number:3
  create-array
     name=order_qty2      size={max_categories}
     field=category:char  field=month_qty:number:3
end-setup
begin-program
  do select_data
  do print_array
  print '-' (+2,1,70) fill
  position (+1)
  do print_array2
end-program
 begin-procedure print_array
  let #entry_cnt = #i
  let #i = 0
  while #i <= #entry_cnt
      let $product  = order_qty.product(#i)
      let #jan      = order_qty.month_qty(#i,0)
      let #feb      = order_qty.month_qty(#i,1)
      let #mar      = order_qty.month_qty(#i,2)
      let #prod_tot = #jan + #feb + #mar
      print $product  (,1,30)
      print #jan      (,32,9) edit 9,999,999
      print #feb      (,42,9) edit 9,999,999
      print #mar      (,52,9) edit 9,999,999
      print #prod_tot (,62,9) edit 9,999,999
      position (+1)
      let #i = #i + 1
```

```
  end-while
end-procedure ! print_array
begin-procedure print_array2
  let #i = 0
  while #i < {max_categories}
      let $category = order_qty2.category(#i)
      let #jan      = order_qty2.month_qty(#i,0)
      let #feb      = order_qty2.month_qty(#i,1)
      let #mar      = order_qty2.month_qty(#i,2)
      let #category_tot = #jan + #feb + #mar
      print $category     (,1,31)
      print #jan          (,32,9) edit 9,999,999
      print #feb          (,42,9) edit 9,999,999
      print #mar          (,52,9) edit 9,999,999
      print #category_tot (,62,9) edit 9,999,999
      position (+1)
      let #jan_total = #jan_total + #jan
      let #feb_total = #feb_total + #feb
      let #mar_total = #mar_total + #mar
      let #i = #i + 1
  end-while
  let #grand_total = #jan_total + #feb_total + #mar_total
  print 'Totals'     (+2,1)
  print #jan_total   (,32,9) edit 9,999,999
  print #feb_total   (,42,9) edit 9,999,999
  print #mar_total   (,52,9) edit 9,999,999
  print #grand_total (,62,9) edit 9,999,999
end-procedure ! print_array2
begin-procedure select_data
  let order_qty2.category(0)='$0-$4.99'
  let order_qty2.category(1)='$5.00-$100.00'
  let order_qty2.category(2)='Over $100'
begin-select
order_date
! the price / price category for the order
c.price &price
  move &price to #price_num
  evaluate #price_num
   when < 5.0
     let #x = 0
     break
  when <= 100.0
     let #x = 1
     break
  when-other
     let #x = 2
     break
  end-evaluate
! The quantity for this order
quantity
  let #j = to_number(datetostr(&order_date,'MM')) - 1
  if #j < 3
      let order_qty2.month_qty(#x,#j) =
                order_qty2.month_qty(#x,#j) + &quantity
  end-if
! the product for this order
description
  if #i = 0 and order_qty.product(#i) = ''
      let order_qty.product(#i) = &description
  end-if
  if order_qty.product(#i) != &description
      let #i = #i + 1
      if #i >= {max_products}
         display 'Error: There are more than {max_products} products'
         stop
      end-if
      let order_qty.product(#i) = &description
  end-if
  if #j < 3
      let order_qty.month_qty(#i,#j) =
                order_qty.month_qty(#i,#j) + &quantity
```



```
  end-if
from  orders a, ordlines b, products c
where a.order_num = b.order_num
and   b.product_code = c.product_code
order by description
end-select
end-procedure ! select_data
begin-heading 5
print $current-date (1,1)
   page-number (1,64) 'Page '
   print 'Order Quantity by Product and Price Category by Month' (2,10)
   print 'Product / Price Category'   (4,1)
   print '  January' (,32)
   print ' February' (,42)
   print '    March' (,52)
   print '    Total' (,62)
   print '-'         (5,1,70) Fill
end-heading
```

The following is the output for program ex8c.sqr:

```
11-JUN-04                                                       Page 1
         Order Quantity by Product and Price Category by Month

Product / Price Category         January  February     March     Total
-------------------------------------------------------------------------
Canisters                              3         0         0         3
Curtain rods                           2         8        18        28
Ginger snaps                           1        10         0        11
Hanging plants                         1        20         0        21
Hookup wire                           16        15         0        31
Hop scotch kits                        2         0         0         2
Modeling clay                          5         0         0         5
New car                                1         9         0        10
Thimble                                7        20         0        27
Thingamajigs                          17         0       120       137
Widgets                                4         0        12        16
Wire rings                             1         0         0         1
-------------------------------------------------------------------------
0-4.99                                28        45        12        85
5.00-100.00                           25        28       138       191
Over 100                               7         9         0        16

Totals                                60        82       150       292
```

SQR arrays are also advantageous in programs that produce charts. With the data for the chart already in an array, presenting this cross-tabular report as a bar chart is easy.

SeeUnderstanding Business Charts .

Chapter 10

Printing Mailing Labels

Understanding Mailing Label Printing

An SQR select paragraph retrieves addresses and prints them on a page.

Sometimes you need to print labels in multiple columns. The page then becomes a matrix of rows and columns of labels. SQR enables you to print in column format with the COLUMNS and NEXT-COLUMN commands in conjunction with the NEXT-LISTING command.

Understanding the Sample Program for Printing Mailing Labels

The following sample program prints mailing labels in a format of 3 columns by 10 rows. It also counts the number of labels that is printed and prints that number in the last sheet of the report.

```
Program ex9a.sqr
#define MAX_LABEL_LINES        10
#define LINES_BETWEEN_LABELS   3
begin-setup
  declare-layout default
    paper-size=(10,11)   left-margin=0.33
  end-declare
end-setup
begin-program
  do mailing_labels
end-program
begin-procedure mailing_labels
  let #label_count = 0
  let #label_lines = 0
  columns 1 29 57  ! enable columns
  alter-printer font=5 point-size=10
begin-select
name       (1,1,30)
addr1      (2,1,30)
city
state
zip
  move &zip to $zip XXXXX-XXXX
   let $last_line = &city || ', ' || &state || ' ' || $zip
  print $last_line (3,1,30)
  next-column at-end=newline
  add 1 to #label_count
  if #current-column = 1
    add 1 to #label_lines
    if #label_lines = {MAX_LABEL_LINES}
      new-page
      let #label_lines = 0
    else
      next-listing no-advance skiplines={LINES_BETWEEN_LABELS}
    end-if
  end-if
from customers
end-select
  use-column 0  ! disable columns
  new-page
```

```
  print 'Labels printed on ' (,1)
  print $current-date ()
  print 'Total labels printed = ' (+1,1)
  print #label_count () edit 9,999,999
end-procedure ! mailing_labels
```

Defining Columns and Rows

The COLUMNS 1 29 57 command defines the starting position for three columns. The first column starts at character position 1, the second at character position 29, and the third at character position 57.

The ex9a.sqr program writes the first address into the first column, the second address into the second column, and the third address into the third column. The program writes the fourth address into the second row of the first column, following the first label. When 10 lines of labels are complete, a new page starts. After the last page of labels is printed, the program prints a summary page showing the number of labels that were printed.

Note the technique for composing the last line of the label. The city, state, and zip columns are moved to string variables. The command `LET $last_line = &city || ', ' || &state || ' ' || $zip` combines the city, state, and zip code, plus appropriate punctuation and spacing, into a string, which it stores in the *$last_line* variable. In this way, city, state, and zip code are printed without unnecessary gaps.

The program defines two counters: #label_count and #label_lines. The first counter, #label_count, counts the total number of labels and prints it on the summary page. The second counter, #label_lines, counts the number of rows of labels that were printed. When the program has printed the number of lines that are defined by `{MAX_LABEL_LINES}`, it starts a new page and resets the #label_lines counter.

After each row of labels, the NEXT-LISTING command redefines the print position for the next row of labels as line 1. NEXT-LISTING skips the specified number of lines (SKIPLINES) from the last line that was printed (NO-ADVANCE) and sets the new position as line 1.

Note the use of the ALTER-PRINTER command. This command changes the font in which the report is printed.

The sample program prints the labels in 10-point Times Roman, which is a proportionally spaced font. In Microsoft Windows, you can use proportionally spaced fonts with any printer that supports fonts or graphics. On other platforms, SQR directly supports HP LaserJet printers and PostScript printers.

In the sample program, the DECLARE-LAYOUT command defines a page width of 10 inches. This width accommodates the printing of the third column, which contains 30 characters and begins at character position 57. SQR assumes a default character grid of 10 characters per inch, which would cause the third column to print beyond the paper edge if this report used the default font. The 10-point Times Roman that is used here, however, condenses the text so that it fits on the page. The page width is set at 10 inches to prevent SQR from treating the third-column print position as an error.

See Printing Charts.

Running the Print Mailing Labels Program

When you print with a proportionally spaced font, you must use a slightly different technique for running the program and viewing the output. If you are using a platform such as UNIX/Linux, specify the printer type with the -PRINTER:*xx* flag. If you are using an HP LaserJet, enter -PRINTER:HP (or -printer:hp). If you are using a PostScript printer, enter -PRINTER:PS (or -printer:ps) on the command line.

For example:

```
sqr ex9a username/password  -printer:hp
```

You can also use the -KEEP command-line flag to produce output in the SQR Portable File format and print it by using SQR Print. You still need to use the -PRINTER:*xx* flag when printing.

See Using the DECLARE-PRINTER Command.

The report produces the output in three columns corresponding to the dimensions of a sheet of mailing label stock. In the preceding example, the report prints the labels from left to right, filling each row of labels before moving down the page.

You can also print the labels from the top down, filling each column before moving to the next column of labels. The code to do this is shown next. The differences between this code and the previous one are shown **like this.** The output is not printed here, but you can run the file and view it using the same procedure that you used for the previous example.

```
Program ex9b.sqr
#define MAX_LABEL_LINES        10
#define LINES_BETWEEN_LABELS   3
begin-setup
  declare-layout default
    paper-size=(10,11)   left-margin=0.33
  end-declare
end-setup
begin-program
  do mailing_labels
end-program
begin-procedure mailing_labels
  let #Label_Count = 0
  let #Label_Lines = 0
  columns 1 29 57  ! enable columns
alter-printer font=5 point-size=10
begin-select
name       (0,1,30)
addr1      (+1,1,30)
city
state
zip
  move &zip to $zip xxxxx-xxxx
  let $last_line = &city || ', ' || &state || ' ' || $zipprint $last_line (+1,1,30)⇒
add 1 to #label_count
  add 1 to #label_lines
  if #label_lines = {MAX_LABEL_LINES}
    next-column goto-top=1 at-end=newpage
    let #label_lines = 0
  else
    position (+1)
    position (+{LINES_BETWEEN_LABELS})
  end-if
from customers
end-select
  use-column 0  ! disable columns
  new-page
```



```
  print 'Labels printed on ' (,1)
  print $current-date ()
  print 'Total labels printed = ' (+1,1)
  print #label_count () edit 9,999,999
end-procedure ! mailing_labels
```

Chapter 11

Creating Form Letters

DOCUMENT Paragraph

To create form letters, use a DOCUMENT paragraph. It starts with a BEGIN-DOCUMENT command and ends with an END-DOCUMENT command. Between these commands, lay out the letter and insert variables where you want data from the database to be inserted. SQR inserts the value of the variable when the document prints. To leave blank lines in a letter, you must explicitly mark them with .b (see the sample program).

Document markers provide another way to add data to a letter. They are special variables whose names begin with @ (the at sign). They mark a location in the document where you place data from areas external to the document paragraph. You can reference document markers defined in DOCUMENT paragraphs in the POSITION command outside the DOCUMENT paragraph to establish the next printing position.

The sample program demonstrates the use of variables and document markers. SQR prints the content of the variable in the position where it is placed in the DOCUMENT paragraph. For example, in the sample program, the customer's name is printed on the first line.

Using a document marker gives you more flexibility in positioning the content of variables. The sample program uses a document marker to position the city, state, and zip code because the city name varies in length and, thus, affects the position of the state name and zip code.

Sample Program for Form Letters

The following simple form letter program, ex10a.sqr, demonstrates the use of document markers:

```
Program ex10a.sqr
begin-program
  do main
end-program
begin-procedure main
begin-select
name
addr1
addr2
city
state
zip
  do write_letter
from customers
order by name
end-select
end-procedure ! main
begin-procedure write_letter
begin-document (1,1)
&name
&addr1
&addr2
```

```
@city_state_zip
.b
.b
                                                    $current-date
Dear Sir or Madam:
.b
     Thank you for your recent purchases from ACME Inc. We would like
     to tell you about our limited-time offer.
     During this month, our entire inventory is marked down by 25%.
     Yes, you can buy your favorite merchandise and save too.
     To place an order simply dial 800-555-ACME.
     Delivery is free too, so don't wait.
.b
.b
                                Sincerely,
                                Clark Axelotle
                                ACME Inc.
end-document
position ()  @city_state_zip
print &city  ()
print ', '   ()
print &state ()
print ' '    ()
print &zip   () edit xxxxx-xxxx
new-page
end-procedure ! write_letter
```

First, SQR performs the main procedure and the select paragraph. Next, it performs the **write_letter** procedure and the document paragraph. The POSITION command sets the position to the appropriate line, which is given by the @city_state_zip marker. The program prints the city, and then it continues printing the other elements to the current position. The state name and zip code automatically print in the correct positions with appropriate punctuation.

The following sample is the output for program ex10a.sqr:

```
John Conway
2837 East Third Street
Greenwich Village
New York, NY 10002-1001


10-MAY-2004

Dear Sir or Madam:

Thank you for your recent purchases from ACME Inc. We would like to tell you
about our limited-time offer.

During this month, our entire inventory is marked down by 25%. Yes, you can
buy your favorite merchandise and save too. To place an order simply dial
800-555-ACME. Delivery is free too, so don't wait.


                                Sincerely,
                                Clark Axelotle
                                ACME Inc.
```

See Adding Graphics.

Chapter 12

Exporting Data to Other Applications

Understanding the Sample Program for Exporting Data

The following sample program creates an export file that you can load into a document such as a spreadsheet or word processing file. The tabs create columns in your spreadsheet or word processing document that correspond to the columns in your database table.

```
Program ex11a.sqr
begin-setup
 ! No margins, wide enough for the widest record
 ! and no page breaks
 declare-layout default
   left-margin=0    top-margin=0
   max_columns=160  formfeed=no
 end-declare
end-setup
begin-program
 do main
end-program
begin-procedure main
encode '<009>' into $sep  ! Separator character is TAB
let $cust_num = 'Customer Number'
let $name = 'Customer Name'
let $addr1 = 'Address Line 1'
let $addr2 = 'Address Line 2'
let $city = 'City'
let $state = 'State'
let $zip = 'Zip Code'
let $phone = 'Phone Number'
let $tot = 'Total'
string $cust_num $name $addr1 $addr2
       $city $state $zip $phone $tot by $sep into $col_hds
print $col_hds (1,1)
new-page
begin-select
cust_num
name
addr1
addr2
city
state
zip
phone
tot
  string &cust_num &name &addr1 &addr2
         &city &state &zip &phone &tot by $sep into $db_cols
  print $db_cols ()
  new-page
from customers
end-select
end-procedure ! main
```

Creating an Export File

The ENCODE command stores the code for the tab character in the *$sep* variable. The code <009> is enclosed within angle brackets to indicate that it is a character that the system does not display. SQR treats it as a character code and sets the variable accordingly. ENCODE is a useful way to place non-alphabetical and nonnumeric characters into variables.

The LET command creates variables for the text strings that are used as column headings in the export file. The STRING command combines these variables in the *$col_hds* variable, with each heading separated by a tab.

The select paragraph uses the STRING command again, this time to combine the records (named as column variables) in the *$db_cols* variable, with each record separated by a tab.

The NEW-PAGE command is used in this example in an unusual way. It causes a new line and carriage return at the end of each record, resetting the line number to 1. The page is not ejected because of the FORMFEED=NO argument in the DECLARE-LAYOUT command. Remember that this report is for exporting, not printing.

You can now load the output file (ex11a.lis) into a spreadsheet or other application.

Chapter 13

Using Graphics

Understanding the Sample Program for Simple Tabular Reports

The following sample program produces a simple tabular report, similar to the one shown in the topic "Selecting Data from the Database."

```
Program ex12a.sqr
begin-setup
 declare-layout default
 end-declare
end-setup
begin-program
 do main
end-program
begin-procedure main
begin-select
name  (,1,30)
city  (,+1,16)
state (,+1,5)
tot   (,+1,11) edit 99999999.99
 next-listing no-advance need=1
 let #grand_total = #grand_total + &tot
from customers
end-select
print '-' (,55,11) fill
print 'Grand Total' (+1,40)
print #grand_total (,55,11) edit 99999999.99
end-procedure ! main
begin-heading 5
 print $current-date (1,1) Edit 'DD-MON-YYYY'
 page-number (1,60) 'Page '
 print 'Name'   (3,1)
 print 'City'   (,32)
 print 'State'  (,49)
 print 'Total'  (,61)
 print '-'      (4,1,65) fill
end-heading
```

The SETUP section contains a DECLARE-LAYOUT command that specifies the default layout without defining any options. The purpose of specifying the default layout is to use its margin settings, which are defined as 1/2 inch. Without DECLARE-LAYOUT, the report would have no margins.

Note the PRINT command with the FILL option. This command produces dashed lines, which is a simple way to draw lines for a report that is printed on a line printer. On a graphical printer, however, you can draw solid lines.

The following is the output for program ex12a.sqr:

```
    06-JUN-04                                                  Page 1

     Name                          City             State     Total
     ---------------------------------------------------------------

     Gregory Stonehaven            Everretsville    OH        39.00
     John Conway                   New York         NY        42.00
     Eliot Richards                Queens           NY        30.00
```

```
Isaiah J Schwartz and Company  Zanesville       OH        33.00
Harold Alexander Fink          Davenport        IN        36.00
Harriet Bailey                 Mamaroneck       NY        21.00
Clair Butterfield              Teaneck          NJ        24.00
Quentin Fields                 Cleveland        OH        27.00
Jerry's Junkyard Specialties   Frogline         NH        12.00
Kate's Out of Date Dress Shop  New York         NY        15.00
Sam Johnson                    Bell Harbor      MI        18.00
Joe Smith and Company          Big Falls        NM         3.00
Corks and Bottles, Inc.        New York         NY         6.00
Harry's Landmark Diner         Miningville      IN         9.00
                                                        ---------
                                        Grand Total      315.00
```

See Adding Graphics.

Adding Graphics

The following sample program includes graphical features, a logo, solid lines, and a change of font, in the heading:

```
Program ex12b.sqr
begin-setup
 declare-layout default
 end-declare
end-setup
begin-program
 do main
end-program
begin-procedure main
begin-select
name  (,1,30)
city  (,+1,16)
state (,+1,5)
tot   (,+1,11) edit 99999999.99
  next-listing no-advance need=1
  let #grand_total = #grand_total + &tot
from customers
end-select
graphic (,55,12) horz-line 20
print 'Grand Total' (+2,40)
print #grand_total (,55,11) Edit 99999999.99
end-procedure ! main
begin-heading 11
 print $current-date (1,1)
 page-number (1,60) 'Page '
 alter-printer point-size=14 font=4 ! switch font
 print 'Name'   (9,1) bold
 print 'City'   (,32) bold
 print 'State'  (,49) bold
 print 'Total'  (,61) bold
 alter-printer point-size=12 font=3 ! restore font
 graphic (9,1,66) horz-line 20
 print-image (1,23)
   type=bmp-file
   image-size=(21,5)
   source='acmelogo.bmp'
end-heading
```

The GRAPHIC command draws solid lines with the HORZ-LINE argument. The line is positioned by using a normal SQR position specifier. Note that the third number in the position specifier is the length of the line, which is given in characters. (The actual width of a character cell is determined by the CHAR-WIDTH or MAX-COLUMNS arguments of DECLARE-LAYOUT.)

The HORZ-LINE argument of the GRAPHIC HORZ-LINE command is the thickness of the line, specified in decipoints (one inch has 720 decipoints). For example, the `graphic (10,1,66) horz-line 20` command specifies a horizontal line following line 10 in the report, starting with position 1 (the left side of the report) and stretching for 66 character positions (at 10 characters per inch, this is 6.6 inches). The thickness of the line is 20 decipoints, which is 1/36 of an inch or about 0.7 mm.

You can also use the GRAPHIC command to draw vertical lines, boxes, and shaded boxes. See the sqrlaser.sqr program in the SAMPLE (or SAMPLEW) subdirectory for an example.

The ALTER-PRINTER command in ex12b.sqr changes the font of the heading. When used a second time, it restores the normal font for the rest of the report. The FONT option selects a font (typeface) that is supported by the printer. The font is specified by number, but the number is printer-specific. On a PostScript printer, for example, font 3 is Courier, font 4 is Helvetica, and font 5 is Times Roman.

The POINT-SIZE option specifies type size in points. You can use a whole number or a fraction (for example, POINT-SIZE=10.5). The following command changes the font to 14-point Helvetica:

```
alter-printer point-size=14 font=4 ! switch font
```

The PRINT-IMAGE command inserts a logo. PRINT-IMAGE is followed by a print position corresponding to the upper-left corner of the image (line 1, column 19 in the sample program). The TYPE option specifies the image file type. In the example, the image is stored in Microsoft Windows bitmap format (bmp file). The size of the image is specified in terms of columns (width) and lines (height). In the example, the image is 30 characters wide (3 inches) and 7 lines high (1-1/6 inches).

In SQR, images are always stored in external files. The format of the image must match that of the printer that you are using. These formats are:

- Microsoft Windows: bmp file images.
- PostScript printer or view: eps file.
- HP LaserJet: hpgl file images.
- HTML output: GIF or JPEG formats (gif file or jpeg file).

The SOURCE option specifies the file name of the image file. In the example, the file is Acmelogo.bmp. The file is assumed to reside in the current directory or in the directory in which SQR is installed (you can place the logo file in either of these places). The file can reside in any directory, however, as long as you specify a full path name for the image file.

The output file now contains graphic language commands. SQR can produce output that is suitable for HP LaserJet printers in a file format that uses the HP PCL language or output that is suitable for PostScript printers in a file format that uses the PostScript language. SQR can also produce printer-independent output files in a special format called SQR Portable Format (SPF).

SQR can create a printer-specific output file (an .lis file) or create the output in portable format (SPF). When you create an .spf file, the name of the image file is copied into it, and the image is processed at print time, when printer-specific output is generated. When you use .spf files, a change in the contents of the image file is reflected in the report the next time you print it or view it. You can create printer-specific output by using SQR or SQR Execute to directly generate an .lis file or by using SQR Print to generate an .lis file from an .spf file.

See Understanding the Sample Program for Listing and Printing Data.

Related Links

SQR Language Reference for PeopleSoft

Sharing Images Among Reports

You can place logos and other images in a report by using only the PRINT-IMAGE command. However, the DECLARE-IMAGE command is useful if you want several programs to share the definition of an image.

The ex12c.sqr program prints a simple form letter. It shows how to print a logo by using the DECLARE-IMAGE and PRINT-IMAGE commands and how to print a signature by using only PRINT-IMAGE.

Because the image is shared among several reports, the DECLARE-IMAGE command is contained in the acme.inc file:

```
File acme.inc
declare-image acme_logo
       type=bmp-file
       image-size=(30,7)
       source='acmelogo.bmp'
end-declare
```

This file declares an image with acme-logo as the name. It specifies the logo that is used in the previous sample program. The declaration includes the type and source file for the image. When the image is printed, you do not need to specify these attributes again.

Multiple programs can share the declaration and include the acme.inc file. If you later need to change an attribute, such as the source, you need to change it in only one place. The image size is specified and provides the default.

To change the size of an image in a particular report, use the IMAGE-SIZE argument of the PRINT-IMAGE command. It overrides the image size that is specified in DECLARE-IMAGE.

```
Program ex12c.sqr
begin-setup
#include 'acme.inc'
end-setup
begin-program
  do main
end-program
begin-procedure main
begin-select
name
addr1
addr2
city
state
zip
phone
  do write_letter
from customers
order by name
end-select
end-procedure ! main
begin-procedure write_letter
move &city to $csz
concat ', ' with $csz
concat &state with $csz
concat ' ' with $csz
move &zip to $zip xxxxx-xxxx
```



```
concat $zip with $csz
move &phone to $phone_no (xxx)bxxx-xxxx   ! Edit phone number.
begin-document (1,1,0)
&name                                              @logo
&addr1
&addr2
$csz
.b
.b
.b
                                                  $current-date
Dear &name
.b
     Thank you for your inquiry regarding Encore, Maestro!!, our revolutionary
teaching system for piano and organ. If you've always wanted to play an
instrument but felt you could never master one, Encore, Maestro!! is made for
you.
.b
     Now anyone who can hum a tune can play one too. Encore, Maestro!! begins
with a step-by-step approach to some of America's favorite songs. You'll learn
the correct keyboarding while hearing the sounds you make through the
headphones provided with the Encore, Maestro!! system. From there, you'll
advance to intricate compositions with dazzling melodic runs. Encore, Maestro!!
can even teach you to improvise your own solos.
.b
     Whether you like classical, jazz, pop, or blues, Encore, Maestro!! is the
music teacher for you.
.b
     A local representative will be calling you at $phone_no
to set up an in-house demonstration, so get ready to play your favorite tunes!!
.b
                                Sincerely,
                                @signature
.b
.b
                                Clark Axelotle
end-document
position () @logo
print-image acme-logo ()
  image-size=(16,4)
position () @signature
print-image ()
  type=bmp-file
  image-size=(12,3)
  source='clark.bmp'
new-page
end-procedure ! write_letter
```

The #INCLUDE command, which is performed at compile time, gets text from another file. In this program, the #INCLUDE 'acme.inc' command includes the code from the acme.inc file.

The document paragraph begins with a BEGIN-DOCUMENT command and ends with an END-DOCUMENT command. It uses variables and document markers to print inside the letter. The program uses variables for the name and address, the date, and the phone number. It uses document markers for the logo and signature.

Document markers are placeholders in the letter. The program uses the @logo and @signature document markers in a POSITION command before printing each image. The document markers make unnecessary specifying the position of these items in the PRINT-IMAGE command. Instead, you print to the current position.

The date is prepared with the *$current-date* reserved variable. It is printed directly in the document paragraph without issuing a PRINT command.

The program uses the CONCAT command to put together the city, state, and zip code. In the document paragraph, variables retain their predefined sizes. A column variable, for example, remains the width

of the column as defined in the database. You can print the date and phone number directly, however, because they occur at the end of a line, without any following text.

Printing Bar Codes

SQR supports a wide variety of bar code types, which you can include in an SQR report.

To create a bar code, use the PRINT-BAR-CODE command. Specify the position of the bar code in an ordinary position qualifier. In separate arguments, specify the bar code type, height, text to be encoded, caption, and optional check sum. For example:

```
print-bar-code (1,1)
  type=1
  height=0.5
  text='01234567890'
  caption='0 12345 67890'
```

Arguments to PRINT-BAR-CODE can be variables or literals.

Related Links

SQR Language Reference for PeopleSoft

Chapter 14

Using Business Charts

Understanding Business Charts

Business charts are useful tools for presenting summary data. SQR provides two commands for creating charts: DECLARE-CHART and PRINT-CHART. It also provides a variety of chart types, including:

- Line
- Pie
- Bar
- Stacked bar
- 100 percent bar
- Overlapped bar
- Floating bar
- Histogram
- Area
- Stacked area
- 100 percent area
- XY scatter plot
- High-low close

You can configure many attributes of SQR charts by activating three-dimensional effects or setting titles and legends. SQR charts are also portable: you can move them from one hardware platform to another.

You can prepare a business chart by using data that is held in an array, just as you would for a cross-tabular report. If you have already written a cross-tabular report, you need to take one additional step to create a chart using the data that is already collected in the array.

See Creating a Chart and Printing Charts.

Creating a Chart

The following sample program (ex8c.sqr) builds on the report that you created in the topic “Creating Cross-Tabular Reports.”. That sample program combined two reports in one program. The following sample program produces two charts that correspond to the two cross-tabular reports.

Here is the code; the lines that were changed or added are shown **like this:**

```
Program ex13a.sqr
#define max-categories 3
#define max-products 100
begin-setup
  create-array
     name=order_qty       size={max-products}
     field=product:char   field=month_qty:number:3
  create-array
     name=order_qty2      size={max-categories}
     field=category:char  field=month_qty:number:3
  declare-chart orders-stacked-bar
    chart-size=(70,30)
    title='Order Quantity'
    legend-title='Month'
    type=stacked-bar
  end-declare ! orders-stacked-bar
end-setup
begin-program
  do select_data
  do print_array
  print '-' (+2,1,70) fill
  position (+1)
  do print_array2new-page
  let $done = 'YES' ! Don't need heading any more
  do print_the_charts
end-program
begin-procedure print_array
  let #entry_cnt = #i
  let #i = 0
  while #i <= #entry_cnt
      let $product  = order_qty.product(#i)
      let #jan      = order_qty.month_qty(#i,0)
      let #feb      = order_qty.month_qty(#i,1)
      let #mar      = order_qty.month_qty(#i,2)
      let #prod_tot = #jan + #feb + #mar
      print $product  (,1,30)
      print #jan      (,32,9) edit 9,999,999
      print #feb      (,42,9) edit 9,999,999
      print #mar      (,52,9) edit 9,999,999
      print #prod_tot (,62,9) edit 9,999,999
      position (+1)
      let #i = #i + 1
  end-while
end-procedure ! print_array
begin-procedure print_array2
  let #i = 0
  while #i < {max_categories}
      let $category = order_qty2.category(#i)
      let #jan      = order_qty2.month_qty(#i,0)
      let #feb      = order_qty2.month_qty(#i,1)
      let #mar      = order_qty2.month_qty(#i,2)
      let #category_tot = #jan + #feb + #mar
      print $category     (,1,31)
      print #jan          (,32,9) edit 9,999,999
      print #feb          (,42,9) edit 9,999,999
      print #mar          (,52,9) edit 9,999,999
      print #category_tot (,62,9) edit 9,999,999
      position (+1)
      let #jan_total = #jan_total + #jan
      let #feb_total = #feb_total + #feb
      let #mar_total = #mar_total + #mar
      let #i = #i + 1
  end-while
  let #grand_total = #jan_total + #feb_total + #mar_total
  print 'Totals'     (+2,1)
  print #jan_total   (,32,9) edit 9,999,999
  print #feb_total   (,42,9) edit 9,999,999
  print #mar_total   (,52,9) edit 9,999,999
```



```
  print #grand_total (,62,9) edit 9,999,999
end-procedure ! print_array2
begin-procedure select_data
  let order_qty2.category(0)='$0-$4.99'
  let order_qty2.category(1)='$5.00-$100.00'
  let order_qty2.category(2)='Over $100'
begin-select
order_date
! the price / price category for the order
c.price &price
  move &price to #price_num
  evaluate #price_num
  when < 5.0
     let #x = 0
     break
  when <= 100.0
     let #x = 1
     break
  when-other
     let #x = 2
     break
  end-evaluate
! The quantity for this order
quantity
   let #j = to_number(datetostr(&order_date,'MM')) - 1
  if #j < 3
      let order_qty2.month_qty(#x,#j) =
                order_qty2.month_qty(#x,#j) + &quantity
  end-if
! the product for this order
description
  if #i = 0 and order_qty.product(#i) = ''
      let order_qty.product(#i) = &description
  end-if
  if order_qty.product(#i) != &description
      let #i = #i + 1
      if #i >= {max_products}
         display 'Error: There are more than {max_products} products'
         stop
      end-if
      let order_qty.product(#i) = &description
  end-if
  if #j < 3
      let order_qty.month_qty(#i,#j) =
                order_qty.month_qty(#i,#j) + &quantity
  end-if
from  orders a, ordlines b, products c
where a.order_num = b.order_num
and   b.product_code = c.product_code
order by description
end-select
end-procedure ! select_data
begin-heading 5if not ($done = 'YES')
   print $current-date (1,1)
   page-number (1,64) 'Page '
   print 'Order Quantity by Product and Price Category by Month' (2,10)
   print 'Product / Price Category'   (4,1)
   print '  January' (,32)
   print ' February' (,42)
   print '    March' (,52)
   print '    Total' (,62)
   Print '-'         (5,1,70) Fill
  end-if
end-heading
 begin-procedure print_the_charts
  print-chart orders-stacked-bar (+2,1)
    data-array=order_qty
    data-array-row-count=12
    data-array-column-count=4
    data-array-column-labels=('Jan','Feb','Mar')
sub-title='By Product By Month'
```



```
  new-page
  print-chart orders-stacked-bar (+2,1)
    data-array=order_qty2
    data-array-row-count=3
    data-array-column-count=4
    data-array-column-labels=('Jan','Feb','Mar')
    sub-title='By Price Category By Month'
end-procedure ! print_the_charts
```

Defining Charts

The two chart sections in the ex13a.sqr program are specified with the DECLARE-CHART command in the SETUP section and are named *orders-stacked-bar*. The width and height of the charts are specified in terms of character cells. The charts that are generated by this program are 70 characters wide, which is 7 inches on a default layout. The height of the charts is 30 lines, which translates to 5 inches at 6 lines per inch. These dimensions define a rectangle that contains the chart. The box that surrounds the chart is drawn by default, but you can disable it by using the qualifier BORDER=NO.

The title is centered at the top of the chart. The text that is generated by LEGEND-TITLE must fit inside the small legend box that precedes the categories, so make this description short. Generally, charts look best when the text items are short. Here is the DECLARE-CHART command:

```
declare-chart orders-stacked-bar
    chart-size=(70,30)
    title='Order Quantity'
    legend-title='Month'
    type=stacked-bar
  end-declare ! orders-stacked-bar
```

The heading prints only on the first page.

Printing Charts

The PRINT-CHART commands are based on the orders-stacked-bar chart that was declared in the preceding section.

```
  print-chart orders-stacked-bar (+2,1)
    data-array=order_qty
    data-array-row-count=12
    data-array-column-count=4
    data-array-column-labels=('Jan','Feb','Mar')
sub-title='By Product By Month'
  new-page
  print-chart orders-stacked-bar (+2,1)
    data-array=order_qty2
    data-array-row-count=3
    data-array-column-count=4
    data-array-column-labels=('Jan','Feb','Mar')
    sub-title='By Price Category By Month'
```

The data source is specified by using the DATA-ARRAY option. The named array has a structure that is specified by the TYPE option. For a stacked-bar chart, the first field in the array gives the names of the categories for the bars. The rest of the fields are series of numbers. In this case, each series corresponds to a month.

The subtitle follows the title and can be used as a second line of the title. A legend labels the series. The DATA-ARRAY-COLUMN-LABELS argument passes these labels. The DATA-ARRAY-ROW-COUNT argument is the number of rows (bars) to chart and DATA-ARRAY-COLUMN-COUNT is the number of fields in the array that the chart uses. The array has four fields: the product (or price category) field and the series that specifies three months.

Running the Program to Create Graphical Reports

When you create a graphical report, you must use a slightly different technique for running the program and viewing the output:

- If you are using a platform such as UNIX/Linux, specify the printer type with the -PRINTER:*xx* flag.
- If you are using an HP LaserJet, enter -PRINTER:HP (or -printer:hp).
- If you are using a PostScript printer, enter -PRINTER:PS (or -printer:ps) in the command line.

  For example:

  ```
  sqr test username/password  -printer:hp
  ```

You can also use the -KEEP command-line flag to produce output in the SQR Portable File format (SPF) and print it using SQR Print. You still must use the -PRINTER:*xx* flag when printing.

Printing Charts and Using the DECLARE-PRINTER Command.

Passing Data to Charts

To pass data to a chart, use the first field for the descriptions of bars (or lines or areas), and then use one or more additional fields with number series. This procedure is common to many chart types, including line, bar, stacked-bar, 100 percent bar, overlapped bar, histogram, area, stacked-area, and 100 percent area. You can omit the first field, and SQR uses cardinal numbers (1, 2, 3, and so on) for the bars. Only text fields are used for these options.

For pie charts, only one series is allowed. Pie charts are a special case because you can specify which segments to explode, or pull away, from the center of a pie. By using a third field in the array, you can have a series of *Y* and *N* values that indicate whether to explode the segment. If *Y* is the value in the first row of the array, then the pie segment that corresponds to the first row is exploded. With pie charts, you cannot omit the first field with the descriptions. Pie charts cannot have more than 12 segments.

Pie charts display a numeric value next to each segment. The description appears in the legend. In addition, SQR displays a percentage next to the numeric value. You can disable this feature by using the qualifier PIE-SEGMENT-PERCENT-DISPLAY=NO.

When data is passed to an *xy* scatter plot or a floating-bar chart, the series are paired. A pair in a floating-bar chart represents the base and height of the bars. A pair in an xy scatter plot represents *x* and *y* coordinates. In an *xy* scatter plot, the first field does not have descriptions. In a floating-bar chart, the first field may have descriptions for the bars. For both types, you can have one or more pairs of series.

Chapter 15

Changing Fonts

Setting Fonts

To select a font in SQR for PeopleSoft, use the DECLARE-PRINTER and ALTER-PRINTER commands. The DECLARE-PRINTER command sets the default font for the entire report. The ALTER-PRINTER command changes the font anywhere in the report, and the change remains in effect until the next ALTER-PRINTER command.

To set a font for an entire report, use ALTER-PRINTER, which is not printer-specific, at the beginning of the program. If you are writing a printer-independent report, the attributes that you set with DECLARE-PRINTER take effect only when you print your report with the printer that you specify with the TYPE argument. To specify a printer at print time, use the -PRINTER:*xx* command-line flag.

Related Links

"ALTER-PRINTER" (PeopleTools 8.58: SQR Language Reference for PeopleSoft)

"DECLARE-PRINTER" (PeopleTools 8.58: SQR Language Reference for PeopleSoft)

Positioning Text

In SQR for PeopleSoft, you position text according to a grid. That grid is set by default to 10 characters per inch and 6 lines per inch, but you can give it another definition by altering the CHAR-WIDTH and LINE-HEIGHT parameters of the DECLARE-LAYOUT command.

Note, however, that character grid and character size function independently of each another. Fonts print in the size that is set by DECLARE-PRINTER or ALTER-PRINTER, not the size that is defined by the grid. A character grid is best used for positioning the first character in a string. It can express the width of a string only in terms of the number of characters that it contains, not in an actual linear measurement, such as inches or picas.

When you use a proportionally spaced font, the number of letters that you print may no longer match the number of character cells that the text actually fills. For example, in the following sample code the word *Proportionally* fills only 9 cells, although it contains 14 letters.

When you print consecutive text strings, the actual position at the end of a string may differ from the position that SQR assumes according to the grid. For this reason, concatenate consecutive pieces of text and print them as one.

For example, do not write code like this:

```
alter-printer font=5    ! select a proportional font
print &first_name ()    ! print first name
print ' ' ()            ! print a space
print &last_name ()     ! print the last name
alter-printer font=3    ! restore the font
```

Instead, write code like this:

```
alter-printer font=5    ! select a proportional font
! concatenate the name
let $full_name = &first_name || ' ' || &last_name
print $full_name ()     ! print the name
alter-printer font=3    ! restore the font
```

The WRAP and CENTER options of the PRINT command also require special consideration when used with proportional fonts. They both calculate the text length based on the character count in the grid, which is not the same as its dimensional width.

Look at the sample program. It contains a list of reminders from the reminders table. It is printed in a mix of fonts: Times Roman in two different sizes plus Helvetica bold.

```
Program ex14a.sqr
begin-setup
 declare-layout default
   paper-size=(10,11)
 end-declare
end-setup
begin-program
 do main
end-program
begin-procedure main
! Set Times Roman as the font for the report
alter-printer font=5 point-size=12
begin-select
remind_date    (,1,20) edit 'DD-MON-YY'
reminder       (,+1) wrap 60 5
  position (+2)
from reminders
end-select
end-procedure ! main
begin-heading 7
 print $current-date    (1,1) Edit 'DD-MON-YYYY'
 page-number (1,60) 'Page '
 ! Use large font for the title
 alter-printer font=5 point-size=24
 print 'Reminder List'      (3,25)
 ! Use Helvetica for the column headings
 alter-printer font=4 point-size=12
 print 'Date' (6,1) bold
 print 'Reminder' (6,22) bold
 graphic (6,1,66) horz-line
 ! Restore the font
 alter-printer font=5 point-size=12
end-heading
```

The report uses the default layout grid of 10 characters per inch and 6 lines per inch both for positioning the text and for setting the length of the solid line.

The font is set at the beginning of the main procedure to font 5, which is Times Roman. The point size is set to 12. In the HEADING section, its size is set to 24 points to print the title.

The column headings are set to 12-point Helvetica with the ALTER-PRINTER FONT=4 POINT-SIZE=12 command. The BOLD option of the PRINT command specifies that they be printed in bold.

A solid line is under the column headings. Note that it is positioned at line 6, the same as the column headings. SQR draws the solid line as an underline. At the end of the HEADING section, the font is restored to Times Roman.

In an SQR program, the report heading is performed after the body. A font change in the heading does not affect the font that is used in the body of the current page, although it changes the font that is used in the

body of subsequent pages. Keep track of your font changes and return fonts to their original settings in the same section in which you change them.

Positioning the title requires careful coding. The CENTER option of the PRINT command does not work because it does not account for the actual size of the text. Instead, position the title by estimating its length. In this case, the title should start 2 1/2 inches from the left margin. The character coordinates are (3,25), which is line 3, character position 25. Remember that the character grid used for positioning assumes 10 characters per inch; therefore, 25 characters is 2 1/2 inches.

Using the WRAP Option

The WRAP option of the PRINT command prints the text of the reminder column. This option wraps text based on a given width, which is 60 characters in the sample program.

The WRAP option works only on the basis of the width that is given in the character grid. It does not depend on the font.

Text that is printed in Times Roman takes about 30 to 50 percent less space than the same text in Courier (the default font, which is a fixed-size font). This means that a column with a nominal width of 44 characters (the width of the reminder column) can actually hold as many as 66 characters when it is printed in the Times Roman font. To be conservative, specify a width of 60.

The other argument of the WRAP option is the maximum number of lines. Because the reminder column in the database is 240 characters wide, at 60 characters per line no more than five lines are needed. Remember, this setting specifies only the maximum number of lines. SQR does not use more lines than necessary.

SQR calculates the maximum number of characters on a line by using the page dimensions in the DECLARE-LAYOUT command (the default is 8 1/2 inches wide). In the sample program, 8 1/2 inches minus the inch that is used in the margins is 7 1/2 inches, or 75 characters at 10 characters per inch. Printing 60 characters starting from position 22 could exceed this maximum and cause an error or undesirable output. To avoid this error, define the page as wider than it actually is. This definition is given by the argument PAPER-SIZE=(10,11) in the DECLARE-LAYOUT command.

Chapter 16

Writing Printer-Independent Reports

Understanding Printer-Independent Reports

To create a printer-independent report, you must write a program that avoids using any characteristics that are unique to a specific printer. Although complete printer independence may be too restrictive, make your report as printer-independent as you can by following these guidelines:

- Ensure that your program is free of the following commands:
    - GRAPHIC FONT (use ALTER-PRINTER instead).
    - PRINTER-INIT, PRINTER-DEINIT, and USE-PRINTER-TYPE (except for using this command to select a printer at runtime, as demonstrated in the sample program that follows).
    - CODE-PRINTER and CODE qualifiers of the PRINT command.
    - DECLARE-PRINTER and PRINT-DIRECT.
    - The SYMBOL-SET argument of the ALTER-PRINTER command.
- Ensure that the report is readable if printed on a line printer. Graphics or solid lines printed with the graphic command are not printed on a line printer. Test your graphical report on a line printer.
- Use only a small set of fonts. Font numbers 3, 4, and 5 and their boldface versions are the same regardless of the type of printer that you use (except for a line printer). Font 3 is Courier, font 4 is Helvetica, and font 5 is Times Roman. Note that on some HP printers, Helvetica may not be available, which would reduce the common fonts to fonts 3 and 5 only.
- Be aware of certain limitations. EPS-file images can be printed only on PostScript printers. HPGL-file images can be printed only on HP LaserJet Series 3 or higher or printers that emulate HP PCL at that level. BMP-file images can be printed using Microsoft Windows only. GIF-file and JPEG-file images are suitable only for HTML output. PRINT-IMAGE and PRINT-CHART may not work with old printers that use PostScript Level 1 or HP LaserJet Series II.

If your report is printer-neutral and does not specify a printer, you can specify the printer at runtime in two ways.

The first method is to use the -PRINTER:*xx* command-line flag, which specifies the output type for your report. Use the following commands:

- -PRINTER:LP for line-printer output.
- -PRINTER:PS for PostScript output.
- -PRINTER:HP for HP LaserJet output.
- -PRINTER:WP for Microsoft Windows output.

- -PRINTER:HT for HTML output.

If you are using the system shell, enter this command in the command line:

```
sqr test username/password -printer:ps
```

Note: Currently, PRINTER:WP sends output to the default Microsoft Windows printer. To specify a non-default Microsoft Windows printer, enter the following command: `-PRINTER:WP:{Printer Name}`. The {Printer Name} is the name assigned to your printer. For example, to send output to a Microsoft Windows printer named NewPrinter, you would use `-PRINTER:WP:NewPrinter`. If your printer name has spaces, enclose the entire command in double quotes.

The second method of specifying the printer type is by using the USE-PRINTER-TYPE command.

See SQR for PeopleSoft Implementation.

Reviewing the Sample Program for Selecting the Printer Type at Runtime

In the following example, the PROGRAM section prompts the user to select the printer type at runtime. The relevant lines are shown **like this:**

```
begin-program
  input $p 'Printer type'    ! Prompt user for printer type
  let $p = lower($p)         ! Convert type to lowercase
  evaluate $p                ! Case statement
  when = 'hp'
  when = 'hplaserjet'        ! HP LaserJet
    use-printer-type hp
    break
  when = 'lp'
  when = 'lineprinter'       ! Line Printeruse-printer-type lp
    break
 when = 'ps'
  when = 'postscript'        ! PostScriptuse-printer-type ps
    break
  when-other
    display 'Invalid printer type.'
    stop
  end-evaluate
  do list_customers
end-program
```

In this code, the INPUT command prompts the user to enter the printer type. Because the USE-PRINTER-TYPE command does not accept a variable as an argument, the EVALUATE command is used to test for the six possible values and set the printer type accordingly.

The EVALUATE command is similar to a switch statement in the C language. It compares a variable to multiple constants and carries out the appropriate code.

Chapter 17

Using Dynamic SQL and Error Checking

Using Variables in SQL

SQL supports the use of variables. A SQL statement containing variables is considered static. When SQR runs a static statement several times, it runs the same statement, even if the values of the variables change. Because SQL allows variables only in places where literals are allowed (such as in WHERE clauses or INSERT statements), the database can parse the statement before the values for the variables are given.

The ex16a.sqr sample program selects customers from a state that the user specifies:

```
Program ex16a.sqr
begin-program
   do list_customers_for_state
end-program
begin-procedure list_customers_for_state
input $state maxlen=2 type=char 'Enter state abbreviation'
let $state = upper($state)
begin-select
name (,1)
   position (+1)
from customers
where state = $state
end-select
end-procedure ! list_customers_for_state
```

Note the use of the *$state* variable in the select paragraph. When you use a variable in a SQL statement in SQR for PeopleSoft, the SQL statement that is sent to the database contains that variable. SQR binds the variable before the SQL is run. In many cases, the database needs to parse the SQL statement only once. The only item that changes between runs of the select paragraph is the value of the variable. This is the most common example of varying a select paragraph.

In the sample program, the INPUT command prompts the user to enter the value of state. The MAXLEN and TYPE arguments verify the input, ensuring that the user enters a string of no more than two characters. If the entry is incorrect, INPUT reprompts.

The sample program converts the contents of the *$state* variable to uppercase, which enables the user to enter the state without worrying about the case. In the example, *state* is uppercase in the database. The sample program shows the LET command that is used with the SQR upper function.

You can let the SQL perform the conversion to uppercase by using `where state = upper($state)` if you are using an Oracle database or by using `where state = ucase($state)` if you are using another database. However, SQR enables you to write database-independent code by moving the use of such SQL extensions to the SQR code.

When you run this program, you must specify one of the states that is included in the sample data for the program to return any records. At the prompt, enter IN, MI, NH, NJ, NM, NY, or OH. If you enter NY (the state where most of the customers in the sample data reside), SQR generates the following output:

```
Output for program ex16a.sqr
John Conway
Eliot Richards
```

```
Harriet Bailey
Kate's Out of Date Dress Shop
Corks and Bottles, Inc.
```

Using Dynamic SQL

You may find it too restrictive to use variables only where literals are allowed. In the following example, the ordering of the records changes based on the user's selection. The program runs the select statement twice. The first time, the first column is called *name* and the second column is called *city*, and the program sorts the records by name with a secondary sort by city. The second time, the first column is the city and the second is name, and the program sorts by city with a secondary sort by name. This is the first select paragraph:

```
select name, city
from customers
order by name, city
```

This is the second select paragraph:

```
select city, name
from customers
order by city, name
```

These statements are different. SQR constructs the statement each time before running it. This technique is called dynamic SQL, and the following sample program illustrates it. To take full advantage of the error-handling procedure, run it with the -CB command-line flag.

```
Program ex16b.sqr
begin-program
   let $col1 = 'name'
   let $col2 = 'city'
   let #pos = 32
   do list_customers_for_state
   position (+1)
   let $col1 = 'city'
   let $col2 = 'name'
   let #pos = 18
   do list_customers_for_state
end-program
begin-procedure give_warning
   display 'Database error occurred'
   display $sql-error
end-procedure ! give_warning
begin-procedure list_customers_for_state
   let $my_order = $col1 || ',' || $col2
begin-select on-error=give_warning
[$col1] &column1=char (,1)
[$col2] &column2=char (,#pos)
   position (+1)
from customers
order by [$my_order]
end-select
end-procedure ! list_customers_for_state
```

When you use variables in an SQL statement in SQR to replace literals and more, you make them *dynamic variables* by enclosing them in square brackets. For example, when you use the *[$my_order]* dynamic variable in the ORDER BY clause of the select paragraph, SQR places the text from the *$my_order* variable in that statement. Each time the statement is run, if the text changes a new statement is compiled and run.

Note: The z/OS operating system does not support square brackets for dynamic variables. Use slashes (/) instead.

Other dynamic variables are *[$col1]* and *[$col2]*. They substitute the names of the columns in the select paragraph. The *&column1* and *&column2* variables are column variables.

You can use dynamic variables to produce reports like this one. The data in the first half of the report is sorted differently from the data in the second half. Also note the **give_warning** error-handling procedure, discussed next.

The following is the output for Program ex16b.sqr:

```
John Conway                    New York
Clair Butterfield              Teaneck
Corks and Bottles, Inc.        New York
Eliot Richards                 Queens
Gregory Stonehaven             Everretsville
Harold Alexander Fink          Davenport
Harriet Bailey                 Mamaroneck
Harry's Landmark Diner         Miningville
Isaiah J Schwartz and Company  Zanesville
Jerry's Junkyard Specialties   Frogline
Joe Smith and Company          Big Falls
Kate's Out of Date Dress Shop  New York
Quentin Fields                 Cleveland
Sam Johnson                    Bell Harbor


Bell Harbor      Sam Johnson
Big Falls        Joe Smith and Company
Cleveland        Quentin Fields
Davenport        Harold Alexander Fink
Everretsville    Gregory Stonehaven
Frogline         Jerry's Junkyard Specialties
Mamaroneck       Harriet Bailey
Miningville      Harry's Landmark Diner
New York         John Conway
New York         Corks and Bottles, Inc.
New York         Kate's Out of Date Dress Shop
Queens           Eliot Richards
Teaneck          Clair Butterfield
Zanesville       Isaiah J Schwartz and Company
```

Using SQL Error Checking

SQR for PeopleSoft checks and reports database errors for SQL statements. When an SQR program is compiled, SQR checks the syntax of the SELECT, UPDATE, INSERT, and DELETE SQL statements in the program. Any SQL syntax error is detected and reported at compile time, before the report is run.

When you use dynamic SQL, SQR cannot check the syntax until runtime. In that case, the content of the dynamic variable is used to construct the SQL statement, which can allow syntax errors to occur in runtime. Errors could occur if the dynamic variables that are selected or used in a WHERE or ORDER BY clause are incorrect.

SQR traps any runtime error, reports the error, and ends the program. To change this default behavior, use the ON-ERROR option of the BEGIN-SELECT or BEGIN-SQL paragraphs:

```
begin-select on-error=give_warning
[$col1] &column1=char (,1)
[$col2] &column2=char (,#pos)
```



```
   position (+1)
from customers
order by [$my_order]
end-select
```

In this sample program, if a database error occurs, SQR invokes a procedure called **give_warning** instead of reporting the problem and ending. Write this procedure like this:

```
begin-procedure give_warning
   display 'Database error occurred'
   display $sql-error
end-procedure ! give_warning
```

This procedure displays the error message but does not stop running the program. Instead, the program continues at the statement immediately following the SQL or SELECT paragraph. Note the use of the *$sql-error* variable, which is a special SQR-reserved variable. It contains the error message text from the database and is automatically set by SQR after a database error occurs.

SQR has a number of reserved, or predefined, variables. For example, the *$sqr-program* variable has the name of the program that is running. The *$username* variable has the user name that was used to sign in to the database. The *#page-count* variable has the page number for the current page.

Using SQL and Substitution Variables

SQR uses the value of a substitution variable to complete the select paragraph at compile time. Because the select paragraph is complete at compile time, SQR can check its syntax before running the program. From this point on, the value of *{my_order}* cannot change and the SQL statement is considered static.

In the following program, the ASK command in the SETUP section prompts the user at compile time. The value that the user enters is placed in a special kind of variable called a *substitution variable.* This variable can be used to substitute any command, argument, or part of a SQL statement at compile time. This example is less common, but it demonstrates the difference between compile-time and runtime substitutions:

```
Program ex16c.sqr
begin-setup
  ask my_order 'Enter the column name to sort by (name or city)'
end-setup
begin-program
   do list_customers_for_state
end-program
begin-procedure give_warning
   display 'Database error occurred'
   display $sql-error
end-procedure ! give_warning
begin-procedure list_customers_for_state
begin-select on-error=give_warning
name (,1)
city (,32)
   position (+1)
from customers
order by {my_order}
end-select
end-procedure ! list_customers_for_state
```

In this case, the ASK command prompts the user for the value of the *{my_order}* substitution variable, which is used to sort the output. If the argument is passed in the command line, no prompt appears. When you run this program, enter name, city, or both (in either order and separated by a comma). The program produces a report that is sorted accordingly.

You can use the ASK command only in the SETUP section. SQR processes ASK commands at compile time before running the program. Therefore, all ASK commands are run before any INPUT command.

INPUT is more flexible than ASK. You can use INPUT inside loops. You can validate the length and type of data input and reprompt if it is not valid. The sample program at the beginning of Using SQL and Substitution Variables topic contains an example of reprompting .

ASK can be more powerful. Substitution variables that are set in an ASK command enable you to modify commands that are normally quite restrictive. The following code shows this technique:

```
begin-setup
   ask hlines 'Number of lines for heading'
end-setup
begin-program
  print 'Hello, World!!' (1,1)
end-program
begin-heading {hlines}
   print 'Report Title' () center
end-heading
```

In this example, the *{hlines}* substitution variable defines the number of lines that the heading will occupy. The BEGIN-HEADING command normally expects a literal and does not allow a runtime variable. When a substitution variable is used with this command, its value is modified at compile time.

See Creating and Running a Sample SQR Program and SQR for PeopleSoft Implementation.

Chapter 18

Using Procedures and Local Variables and Passing Arguments

Using Procedures

The code example in this section shows a procedure that spells out a number. The sample program for printing checks uses this procedure. When printing checks, you normally need to spell out the dollar amount.

In the spell.inc code example, the assumption is that the checks are preprinted and that the program has to print only items such as the date, name, and amount.

SQR procedures that contain variables that are visible throughout the program are called global procedures. These procedures can also directly reference any program variable.

In contrast, procedures that take arguments, such as the **spell_number** procedure in the check printing sample program in this section, are local procedures. In SQR for PeopleSoft, any procedure that takes arguments is automatically considered local.

Variables that are introduced in a local procedure are readable only inside the spell.inc procedure. This useful feature avoids name collisions. The **spell_number** procedure is in an include file because you may want to use it for other reports.

Using Local Variables

When you create library procedures that can be used in many programs, make them local. Then, if a program has a variable with the same name as a variable that is used in the procedure, a collision will not occur. SQR treats the two variables as separate.

Declare a procedure as local even if it does not take any arguments. To do this, place the LOCAL keyword following the procedure name in the BEGIN-PROCEDURE command.

To reference a global variable from a local procedure, insert an underscore between the prefix character (#, $, or &) and the variable name. Use the same technique to reference reserved variables , such as *#current-line.* These variables are always global so that you can reference them from a local procedure.

SQR supports recursive procedure calls, but it maintains only one copy of a local variable. A procedure does not allocate new instances of the local variables on a stack, as C or Pascal would.

Passing Arguments

Procedure arguments are treated as local variables. Arguments can be numeric, date, or text variables or strings. If an argument is preceded with a colon, its value is passed back to the calling procedure.

In the following code example, `spell_number` takes two arguments. The first argument is the check amount. This argument is a number, and the program passes it to the procedure. The procedure does not need to pass it back.

The second argument is the result that the procedure passes back to the calling program. We precede this variable with a colon, which means that the value of this argument is copied back at the end of the procedure. The colon is used only when the argument is declared in the BEGIN-PROCEDURE command.

Look at the following sample program. It is not a complete program, but it is the **spell_number** procedure, which is stored in the spell.inc file. The check printing sample program includes this code by using an #INCLUDE command.

```
File spell.inc
begin-procedure spell_number(#num,:$str)
   let $str = ''
   ! break the number to it's 3-digit parts
   let #trillions  = floor(#num / 1000000000000)
   let #billions   = mod(floor(#num / 1000000000),1000)
   let #millions   = mod(floor(#num / 1000000),1000)
   let #thousands  = mod(floor(#num / 1000),1000)
   let #ones       = mod(floor(#num),1000)
   ! spell each 3-digit part
   do spell_3digit(#trillions,'trillion',$str)
   do spell_3digit(#billions,'billion',$str)
   do spell_3digit(#millions,'million',$str)
   do spell_3digit(#thousands,'thousand',$str)
   do spell_3digit(#ones,'',$str)
end-procedure ! spell_number
begin-procedure spell_3digit(#num,$part_name,:$str)
   let #hundreds = floor(#num / 100)
   let #rest     = mod(#num,100)
   if #hundreds
      do spell_digit(#hundreds,$str)
      concat 'hundred ' with $str
   end-if
   if #rest
   do spell_2digit(#rest,$str)
   end-if
   if #hundreds or #rest
      if $part_name != ''
         concat $part_name with $str
          concat ' ' with $str
      end-if
   end-if
end-procedure ! spell_3digit
begin-procedure spell_2digit(#num,:$str)
   let #tens     = floor(#num / 10)
   let #ones     = mod(#num,10)
   if #num < 20 and #num > 9
      evaluate #num
       when = 10
         concat 'ten ' with $str
         break
       when = 11
         concat 'eleven ' with $str
         break
   when = 12
         concat 'twelve ' with $str
         break
```



```
        when = 13
           concat 'thirteen ' with $str
           break
         when = 14
           concat 'fourteen ' with $str
           break
       when = 15
           concat 'fifteen ' with $str
           break
         when = 16
           concat 'sixteen ' with $str
           break
         when = 17
           concat 'seventeen ' with $str
           break
         when = 18
           concat 'eighteen ' with $str
           break
         when = 19
           concat 'nineteen ' with $str
           break
        end-evaluate
    else
       evaluate #tens
        when = 2
          concat 'twenty' with $str
          break
        when = 3
          concat 'thirty' with $str
          break
        when = 4
          concat 'forty' with $str
          break
       when = 5
          concat 'fifty' with $str
          break
        when = 6
          concat 'sixty' with $str
           break
       when = 7
          concat 'seventy' with $str
          break
        when = 8
          concat 'eighty' with $str
          break
        when = 9
          concat 'ninety' with $str
          break
        end-evaluate
      if #num > 20
           if #ones
              concat '-' with $str
           else
              concat ' ' with $str
           end-if
        end-if
        if #ones
           do spell_digit(#ones,$str)
        end-if
    end-if
end-procedure ! spell_2digit
begin-procedure spell_digit(#num,:$str)
    evaluate #num
       when = 1
          concat 'one ' with $str
          break
       when = 2
          concat 'two ' with $str
          break
       when = 3
          concat 'three ' with $str
```



```
         break
      when = 4
         concat 'four ' with $str
         break
      when = 5
         concat 'five ' with $str
         break
      when = 6
         concat 'six ' with $str
         break
      when = 7
         concat 'seven ' with $str
         break
      when = 8
         concat 'eight ' with $str
         break
      when = 9
         concat 'nine ' with $str
         break
   end-evaluate
end-procedure ! spell_digit
```

The result argument is reset in the procedure because the program begins with an empty string and keeps concatenating the parts of the number to it. The program supports numbers up to 999 trillion only.

The number is divided into its three-digit parts: trillions, billions, millions, thousands, and ones. Another procedure spells out the three-digit numbers, such as one hundred twelve. Note that the word *and* is inserted only between dollars and cents, but not between three-digit parts. This format is common for check printing in dollars.

Note the use of math functions, such as floor and mod. SQR for PeopleSoft has a large set of functions that can be used in expressions. These functions are listed and described under the LET command.

See "SQR Commands" (PeopleTools 8.58: SQR Language Reference for PeopleSoft).

The series of EVALUATE commands in the number spelling procedures are used to correlate the numbers that are stored in the variables with the strings that are used to spell them out.

This is the sample program that prints checks:

```
Program ex17a.sqr
#include 'spell.inc'
begin-setup
  declare-layout default
  end-declare
end-setup
begin-program
  do main
end-program
begin-procedure main
  alter-printer font=5 point-size=15
begin-select
name                                &name
sum(d.price * c.quantity) * 0.10    &refund
  do print_check(&refund)
from  customers a, orders b,
      ordlines c, products d
  where a.cust_num = b.cust_num
  and   b.order_num = c.order_num
  and   c.product_code = d.product_code
 group by name
 having sum(d.price * c.quantity) * 0.10 >= 0.01
end-select
end-procedure ! main
begin-procedure print_check(#amount)
  print $_current-date (3,45) edit 'DD-Mon-YYYY'
```



```
   print &_name (8,12)
  move #amount to $display_amt 9,999,990.99
  ! enclose number with asterisks for security
  let $display_amt = '**' || ltrim($display_amt,' ') || '**'
  print $display_amt (8,58)
  if #amount < 1.00
    let $spelled_amount = 'Zero dollars '
  else
    do spell_number(#amount,$spelled_amount)
    let #len = length($spelled_amount)
    ! Change the first letter to uppercase
    let $spelled_amount = upper(substr($spelled_amount,1,1))
                || substr($spelled_amount,2,#len - 1)
    concat 'dollars ' with $spelled_amount
  end-if
  let #cents = round(mod(#amount,1) * 100, 0)
  let $cents_amount = 'and ' || edit(#cents,'00') || ' cents'
  concat $cents_amount with $spelled_amount
  print $spelled_amount (12,12)
  print 'Rebate'        (16,12)
  print ' ' (20)
  next-listing need=20
end-procedure ! print_check
```

The **main** procedure starts by setting the font to 15-point Times Roman. The select paragraph is a join of several tables. (A join is created when you select data from more than one database table in the same select paragraph.) The customers table has the customer's name. The program joins it with the orders and ordlines tables to get the customer's order details. It also joins with the products table for the price.

The following expression adds up all of the customer's purchases and calculates a 10 percent rebate:

```
sum(d.price * c.quantity) * 0.10
```

The statement groups the records by the customer name, one check per customer, using the following clause:

```
group by name
having sum(d.price * c.quantity) * 0.10 >= 0.01
```

The having clause eliminates checks for less than 1 cent.

The **print_check** procedure is a local procedure. Note the way that it references the date and customer name with *&_current-date* and *&_name,* respectively.

See SQR Language Reference for PeopleSoft.

Chapter 19

Creating Multiple Reports from One Program

Understanding How to Create Multiple Reports

You can create multiple reports based on common data, selecting the database records only once and creating different reports simultaneously. The alternative—writing separate programs for the different reports—would require you to perform a separate database query for each report. Repeated queries are costly because database operations are often the most resource consuming or time consuming part of creating a report. Creating multiple reports from one program can save a significant amount of processing time.

Understanding the Sample Program for Multiple Reports

The following sample program, ex18a.sqr, shows how SQR for PeopleSoft enables you to write multiple reports with different layouts and different heading and footing sections. The sample program prints three reports: the labels from the "Printing Mailing Labels" topic, the form letter from the "Creating Form Letters" topic, and the listing report from the "Selecting Data from the Database" topic. All three reports are based on the same data.

```
Program ex18a.sqr
#define MAX_LABEL_LINES        10
#define LINES_BETWEEN_LABELS   3
begin-setup
  declare-layout labels
    paper-size=(10,11)   left-margin=0.33
  end-declare
  declare-layout form_letter
  end-declare
  declare-layout listing
  end-declare
  declare-report labels
    layout=labels
  end-declare
  declare-report form_letter
    layout=form_letter
  end-declare
  declare-report listing
    layout=listing
   end-declare
end-setup
begin-program
  do main
end-program
begin-procedure main
  do init_mailing_labels
begin-select
name
addr1
addr2
city
```

```
state
zip
  move &zip to $zip xxxxx-xxxx
phone
  do print_label
  do print_letter
  do print_listing
from customers
end-select
  do end_mailing_labels
end-procedure ! main
begin-procedure init_mailing_labels
  let #label_count = 0
  let #label_lines = 0
  use-report labels
  columns 1 29 57  ! enable columns
  alter-printer font=5 point-size=10
end-procedure ! init_mailing_labels
begin-procedure print_label
  use-report labels
  print &name   (1,1,30)
  print &addr1  (2,1,30)
  let $last_line = &city || ', ' || &state || ' ' || $zip
  print $last_line (3,1,30)
  next-column at-end=newline
  add 1 to #label_count
  if #current-column = 1
    add 1 to #label_lines
    if #label_lines = {MAX_LABEL_LINES}
      new-page
      let #label_lines = 0
    else
      next-listing no-advance skiplines={LINES_BETWEEN_LABELS}
    end-if
  end-if
end-procedure ! print_label
begin-procedure end_mailing_labels
  use-report labels
  use-column 0  ! disable columns
  new-page
  print 'Labels printed on ' (,1)
  print $current-date ()
  print 'Total labels printed = ' (+1,1)
  print #label_count () edit 9,999,999
end-procedure ! end_mailing_labels
begin-procedure print_letter
use-report form_letter
begin-document (1,1)
&name
&addr1
&addr2
@city_state_zip
.b
.b
                                                    $current-date
Dear Sir or Madam:
.b
    Thank you for your recent purchases from ACME Inc. We would
like to tell you about our limited time offer. During this month,
our entire inventory is marked down by 25%. Yes, you can buy your
favorite merchandise and save too.
    To place an order simply dial 800-555-ACME.
    Delivery is free too, so don't wait.
.b
.b
                                Sincerely,
                                Clark Axelotle
                                ACME Inc.
end-document
position () @city_state_zip
print &city  ()
```

```
print ', '   ()
print &state ()
print ' '    ()
move &zip to $zip xxxxx-xxxx
print $zip   ()
new-page
end-procedure ! print_letter
begin-heading 4 for-reports=(listing)
print 'Customer Listing' (1) center
   print 'Name' (3,1)
   print 'City' (,32)
   print 'State' (,49)
   print 'Phone' (,55)
end-heading
begin-footing 1 for-reports=(listing)
   ! Print "Page n of m" in the footing
   page-number (1,1) 'Page '
   last-page   () ' of '
end-footing
begin-procedure print_listing
  use-report listing
  print &name (,1)
  print &city (,32)
  print &state (,49)
  print &phone (,55)
  position (+1)
end-procedure ! print_listing
```

The SETUP section defines three layouts and three different reports that use these layouts. The labels report requires a layout that is different from the default. The other two reports use a layout that is identical to the default layout. You can save the last layout declaration and use the form letter layout for the listing. However, unless a logical reason exists why the two layouts should be the same, you should keep separate layouts. The name of the layout indicates which report uses it.

The main procedure performs the Select command. The <command> is performed only once and includes all of the columns for all of the reports. The phone column is used only in the listing report, and the addr2 column is used only in the form letter report. The other columns are used in more than one report.

For each record that is selected, three procedures are run. Each procedure processes one record for its corresponding report. The **print_label** procedure prints one label, the **print_letter** procedure prints one letter, and the **print_listing** procedure prints one line in the listing report. Each procedure begins by setting the SQR printing context to its corresponding report. SQR sets the printing context with the USE-REPORT command.

Defining Heading and Footing Sections

SQR enables you to define HEADING and FOOTING sections for each report. This sample program defines only the heading and footing sections for the listing report because the other two reports do not use them. The FOR-REPORTS option of the BEGIN-HEADING and BEGIN-FOOTING commands specifies the report name. The parentheses are required. The USE-REPORT command is not needed in the heading or footing sections. The report is implied by the FOR-REPORTS option.

Defining Program Output

Most of the code for ex18a.sqr is taken from ex9a.sqr, ex10a.sqr, and ex3a.sqr. Because this program creates output with proportional fonts, you must run it with the -KEEP or -PRINTER:*xx* command-line flags.

When you run ex18a.sqr, you get three output files that match the output files for ex9a, ex10a, and ex3a, respectively. These output files have the names ex18a.lis (labels), ex18a.l01 (form letter), and ex18a.l02 (customer listing). If you specify -KEEP, the output files are named ex18a.spf, ex18a.s01, and ex18a.s02, respectively.

Related Links

Understanding SQR Pages
Understanding Mailing Label Printing
Sample Program for Form Letters

Chapter 20

Using Additional SQL Statements with SQR

Using SQL Statements in SQR

Although SELECT may be the most common SQL statement, you can also perform other SQL commands in SQR. Here are a few examples:

- If the program prints important documents such as checks, tickets, or invoices, you may need to update the database to indicate that the document was printed.

  You can do this in SQR with a SQL UPDATE statement.

- You can use SQR to load data into the database.

  SQR can read and write external files and construct records. SQR can also insert these records into the database by using a SQL INSERT statement.

- To hold intermediate results in a temporary database table, you can create two SQL paragraphs in the SQR program (CREATE TABLE and DROP TABLE) to create this table at the beginning of the program and drop the table at the end.

These are only a few examples. SQR can perform any SQL statement, and this feature is used often.

Using the BEGIN-SQL Paragraph

A SQL statement other than a select statement must use the BEGIN-SQL paragraph.

The following sample program loads data from an external file into the database. It demonstrates two important features of SQR: handling external files and performing database inserts. This sample program loads the tab-delimited file that is created by the program ex11a.sqr:

```
Program ex19a.sqr
begin-setup
  begin-sql  on-error=skip ! table may already exist
    create table customers_ext (
      cust_num int not null,
      name     varchar (30),
      addr1    varchar (30),
      addr2    varchar (30),
      city     varchar (16),
      state    varchar (2),
      zip      varchar (10),
      phone    varchar (10),
      tot      int
      )
  end-sql
end-setup

begin-program
  do main
end-program
```

```
begin-procedure main

  encode '<009>' into $sep
  open 'ex11a.lis' as 1 for-reading record=160:vary
  read 1 into $rec:160  ! skip the first record, column headings
  while 1
    read 1 into $rec:160
    if #end-file
      break
    end-if
    unstring $rec by $sep into $cust_num $name
        $addr1 $addr2 $city $state $zip $phone $tot
    move $cust_num to #cust_num
    move $tot to #tot
    begin-sql
       insert into customers_ext (cust_num, name,
         addr1, addr2, city, state, zip, phone, tot)
      values
         (#cust_num, $name, $addr1, $addr2, $city,
          $state, $zip, $phone, #tot)
    end-sql
  end-while

  begin-sql
    commit
  end-sql

  close 1
end-procedure! main
```

The sample program begins by creating the customers_ext table. If the table already exists, you receive an error message. To ignore this error message, use the ON-ERROR=SKIP option.

The program reads the records from the file and inserts each record into the database by using an insert statement inside a BEGIN-SQL paragraph. The input file format is one record per line, with each field separated by the separator character. When the end of the file is encountered (if #end-file), the program branches out of the loop. Note that *#end-file* is an SQR-reserved variable.

The final step is to commit the changes to the database and close the file. You do this with a SQL COMMIT statement inside a BEGIN-SQL paragraph. Alternatively, you can use the SQR COMMIT command. For Oracle databases, use the SQR COMMIT command.

The code may be database-specific.

See Improving SQL Performance with Dynamic SQL.

See SQR Language Reference for PeopleSoft.

Chapter 21

Working with Dates

Understanding Dates and Date Arithmetic

SQR has powerful capabilities in date arithmetic, editing, and manipulation. A date can be represented as a character string or in an internal format by using the SQR date data type.

The date data type enables you to store dates in the range of January 1, 4712 BC to December 31, 9999 AD. It also stores the time of day with the precision of a microsecond. The internal date representation always keeps the year as a four-digit value. Keep dates with four-digit year values (instead of truncating to two digits) to avoid date problems at the turn of the century.

You can obtain date values:

- By selecting a date column from the database.
- By using INPUT to get a date from the user.
- By referencing or printing the *$current-date* reserved variable.
- By using the SQR date functions dateadd, datediff, datenow, or strtodate.
- By declaring a date variable using the DECLARE-VARIABLE command.

For most applications, you do not need to declare date variables. Date variables are discussed later in the section.

Many applications require date calculations. You may need to add or subtract a number of days from a given date, subtract one date from another to find a time difference, or compare dates to determine whether one date is later, earlier, or the same as another date. SQR enables you to perform these calculations in your program.

Many databases enable you to perform date calculations in SQL, but doing so can be difficult if you are trying to write portable code because the syntax varies among databases. Instead, perform those calculations in SQR; your programs will be portable because they will not rely on a particular SQL syntax.

The dateadd function adds or subtracts a number of specified time units from a given date. The datediff function returns the difference between two specified dates in the time units that you specify: years, quarters, months, weeks, days, hours, minutes, or seconds. Fractions are allowed; you can add 2.5 days to a given date. Conversion among time units is also allowed; you can add, subtract, or compare dates by using days and state the difference by using weeks.

The datenow function returns the current local date and time. In addition, SQR provides a reserved date variable, *$current-date,* which is automatically initialized with the local date and time at the beginning of the program.

You can compare dates by using the usual operators (< , =, or >) in an expression. The datetostr function converts a date to a string. The strtodate function converts a string to a date.

The following sample program uses functions to add 30 days to the invoice date and to compare it to the current date:

```
begin-select
order_num     (,1)
invoice_date
   if dateadd(&invoice_date,'day',30) < datenow()
      print 'Past Due Order' (,12)
   else
      print 'Current Order'  (,12)
   end-if
   position (+1)
end-select
```

This code example uses the dateadd and datenow functions to compare dates. The dateadd function adds 30 days to the invoice date `(&invoice_date)`. The resulting date is then compared with the current date, which is returned by datenow. If the invoice is older than 30 days, the program prints the *Past Due Order* string. If the invoice is 30 days old or less, the program prints the *Current Order* string.

To subtract a given number of days from a date, use the dateadd function with a negative argument. This technique is demonstrated in the next code example. In this example, the IF condition compares the invoice date with the date of 30 days before today. The condition is equivalent to that of the previous code example.

```
   if &invoice_date < dateadd(datenow(),'day',-30)
```

You can also write this condition as follows by using the datediff function. Note that the comparison is now a simple numeric comparison, not a date comparison:

```
   if datediff(datenow(),&invoice_date,'day') > 30
```

All three IF statements are equivalent, and they demonstrate the flexibility that is provided by these functions.

Here is another technique for comparing dates:

```
begin-select
order_date
   if &order_date > strtodate('3/1/2004','dd/mm/yyyy')
     print 'Current Order' ()
   else
     print 'Past Due Order' ()
   end-if
from orders
end-select
```

The IF statement has a date column on the left side and the strtodate function on the right side. The strtodate function returns a date type, which is compared with the &order_date column. When the order date is later than January 3, 2004, the condition is satisfied. If the date includes the time of day, the comparison is satisfied for orders of January 3, 2004, with a time of day greater than 00:00.

In the next code example, the date is truncated to remove the time-of-day portion of a date:

```
if strtodate(datetostr(&order_date,'dd/mm/yyyy'),'dd/mm/yyyy') >
       strtodate('3/1/2004','dd/mm/yyyy')
```

In this code example, the datetostr function converts the order date to a string that stores only the day, month, and year. The strtodate function then converts this value back into a date. With these two

conversions, the time-of-day portion of the order date is omitted. Now when it is compared with January 3, 2004, only dates that are of January 4 or later satisfy the condition.

Using Literal Date Formats

SQR enables you to specify date constants and date values in a special format that is recognized without the use of an edit mask. This is called the literal date format. For example, you can use a value in this format in the strtodate function without the use of an edit mask. This format is independent of any specific database or language preference.

The literal date format is SYYYYMMDD[HH24[MI[SS[NNNNNN]]]]. The first *S* in this format represents an optional minus sign. If preceded with a minus sign, the string represents a BC date. The digits that follow represent year, month, day, hours, minutes, seconds, and microseconds.

Note: The literal date format assumes a 24-hour clock.

You can omit one or more time elements from the right part of the format. A default is assumed for the missing elements. Here are some code examples:

```
let $a = strtodate('20040409')
let $a = strtodate('20040409152000')
```

The first LET statement assigns the date of April 9, 2004 to the *$a* variable. The default time portion is 00:00. The second LET statement assigns 3:20 in the afternoon of April 9, 2004 to *$a*. The outputs (when printed with the 'DD-MON-YYYY HH:MI AM' edit mask) are, respectively:

```
09-APR-2004 12:00 AM
09-APR-2004 03:20 PM
```

You can also specify a date format with the SQR_DB_DATE_FORMAT environment variable. You can specify this as an environment variable or in the pssqr.ini file.

Related Links

Understanding the SQR Command Line

Using String-to-Date Conversions

If you convert a string variable or constant to a date variable without specifying an edit mask that identifies the format of the string, SQR applies a date format. This implicit conversion takes place with these commands:

- MOVE.
- The strtodate function.
- The DISPLAY, PRINT, or SHOW commands when used to format a string variable as a date.

SQR attempts to apply date formats in this order:

1. The format specified in SQR_DB_DATE_FORMAT.

2. The database-dependent format.
3. The SYYYYMMDD[HH24[MI[SS[NNNNNN]]]] literal date format.

Using Date-to-String Conversions

If you convert a date variable to a string without specifying an edit mask, SQR applies a date format. The conversion takes place with:

- The MOVE command.
- The datetostr function.
- The DISPLAY, PRINT, or SHOW commands when used to output a date variable.

SQR attempts to apply date formats in this order:

1. The format specified in SQR_DB_DATE_FORMAT.
2. The database-dependent format.

Related Links

SQR Language Reference for PeopleSoft

Using Dates with the INPUT Command

The INPUT command also supports dates. You can load a date into a date or string variable. For string variables, use the TYPE=DATE qualifier. Specify a format for the date. Here is a code example:

```
input $start_date 'Enter starting date'  type=date  format='dd/mm/yyyy'
```

In this example, the user is prompted with *Enter starting date:* (the colon is automatically added). The user then enters the value, which is validated as a date by using the dd/mm/yyyy format. The value is loaded into the *$start_date* variable.

Using Date Edit Masks

When you print dates, you can format them with an edit mask. For example:

```
print &order_date () edit 'Month dd, YYYY'
```

This command prints the order date in the specified format. The name of the order date month is printed, followed by the day of the month, a comma, and four-digit year. SQR for PeopleSoft provides an extensive set of date edit masks.

See "PRINT" (PeopleTools 8.58: SQR Language Reference for PeopleSoft).

If the value of the date value being edited is March 14, 2004 at 9:35 in the morning, the edit masks produce the following results:

Edit Mask	*Result*	*Notes*
dd/mm/yyyy	14/03/2004	NA
DD-MON-YYYY	14-MAR-2004	NA
'Month dd, YYYY.'	March 14, 2004.	An edit mask containing blank space must be enclosed in single quotes.
MONTH-YYYY	MARCH-2004	The name of the month in uppercase, followed by the 4-digit year.
HH:MI	09:35	NA
'HH:MI AM'	09:35 AM	Meridian indicators. An edit mask containing blank space must be enclosed in single quotes.
YYYYMMDD	20040314	NA
DD.MM.YY	14.03.99	NA
Mon	Mar	The abbreviated name of the month.
Day	Thursday	The day of the week.
DY	THU	An abbreviation for the day of the week.
Q	1	Quarter.
WW	11	The week of the year.
W	2	The week of the month.
DDD	74	The day of the year.
DD	14	The day of the month (1–31).
D	3	The day of the week (Sunday is 1).
EY	Please see below	The Japanese imperial era (Meiji, Taisho, Showa, Heisei).
ER	16	The year in Japanese imperial era.

The result for EY is:

Japanese Imperial Era

Note: The MON, MONTH, DAY, DY, AM, PM, BC, AD, ER, EY, and RM masks are case-sensitive and follow the case of the mask that is entered. For example, if the month is January, the Mon mask yields *Jan* and MON yields *JAN.* All other masks are case-insensitive and can be entered in either uppercase or lowercase.

If the edit mask contains other text, it is also printed. For example:

```
print &order_date () edit 'As of Month dd, YYYY'
```

This command prints the *As of March 14, 2004* string if the order date is March 14, 2004. Because the words *As of* are not recognized as date mask elements, they are printed.

A backslash forces the character that follows into the output. This technique is useful to print text that would otherwise be recognized as a date mask element. For example, a mask of *The \mo\nth is Month* results in *The month is March* as an output string. Without the backslashes, the output string would be *The march is March.* The second backslash is needed because *n* is a valid date edit mask element.

In some cases, combining date edit mask elements can result in ambiguity. One example is the 'DDDD' mask, which could be interpreted as various combinations of DDD (day of year), DD (day of month), and D (day of week). To resolve such ambiguity, use a vertical bar as a delimiter between format elements. For example, DDD followed by D can be written as DDD|D.

In addition, national language support is provided for the following masks: MON, MONTH, DAY, DY, AM, PM, BC, and AD.

Related Links

SQR Language Reference for PeopleSoft

Declaring Date Variables

To hold date values in your program, use date variables. Like string variables, date variables are prefixed with a dollar sign ($). You must explicitly declare date variables by using the DECLARE-VARIABLE command.

Date variables are useful for holding results of date calculations. For example:

```
begin-setup
   declare-variable
      date $c
   end-declare
end-setup
...
let $c = strtodate('March 1, 2004 12:00','Month dd, yyyy hh:mi')
print $c () edit 'dd/mm/yyyy'
```

In this code example, *$c* is declared as a date variable. Later, it is assigned the value of noon on March 1, 2004. The *$c* variable is then printed with the dd/mm/yyyy edit mask, which yields *01/03/2004.*

Date variables can be initialized with date literals as shown in this example:

```
begin-setup
   declare-variable
      date $c
   end-declare
end-setup
...
let $c = '20040409152000'
```

The LET statement assigns 3:20 in the afternoon of April 9, 2004 to *$c.*

Chapter 22

Using National Language Support

Understanding Locales

National Language Support (NLS) is provided through the concept of locales. A *locale* is a set of local preferences for language, currency, and the presentation of dates and numbers. For example, one locale may use English, dollar currency, dates in dd/mm/yy format, numbers with commas separating the thousands, and a period for the decimal place.

A locale contains:

- Default edit masks for number, money, and date.

  Use these edit masks to specify the NUMBER, MONEY (for currency), and DATE keywords, respectively. You can specify these keywords in the INPUT, MOVE, DISPLAY, SHOW, and PRINT commands.

- Settings for currency symbol, thousands separator, decimal separator, date separator, and time separator.
- Settings for not applicable (NA), a.m., p.m., BC, and AD in the language of the locale.
- Settings for names of the days of the week and names of the months in the language of the locale.
- Settings for how to process lowercase and uppercase editing of day and month names.

Starting with PeopleTools 8.54, you can generate region-specific SQR reports without personalizing or modifying the SQR scripts.

Selecting Locales

SQR provides predefined locales such as US-English, UK-English, German, French, and Spanish. You can define additional locales by editing any .ini file.

With the ALTER-LOCALE command, you can select a locale at the beginning of the program or anywhere else. Different parts of a program can use different locales.

Select a locale with a command such as this:

```
alter-locale locale = 'German'
```

Defining a Default Locale

You can define a default locale in any .ini file. Most or all of your programs can use the same locale, and specifying the default locale makes specifying the locale in every program unnecessary.

When you install SQR, the default locale is set to the reserved locale called *System*. *System* is not an actual locale. It defines the behavior of older versions of SQR before NLS was added. The preferences in the system locale are hard-coded in the product and cannot be set or defined in the pssqr.ini; however, you can alter system settings at runtime by using ALTER-LOCALE. The date preferences depend on the database that you are using. Therefore, date format preferences in the system locale are different for every database that you use with SQR.

Note: If you are running SQR outside of the PeopleSoft Process Scheduler, the PS_HOME environment variable must be set to a proper PeopleSoft installation.

Different sites can have different locales as the default. For example, an office in Paris might use the French locale, and an office in London might use the UK-English locale. To adapt a program to any location, use the default locale. The program automatically uses the local preferences, which are specified in the pssqr.ini file of the machine on which it is run. For example, you can print the number 5120 by using the following command:

```
print #invoice_total () edit '9,999,999.99'
```

The setting of the default locale in the pssqr.ini file controls the format. In London, the result might be 5,120.00 and in Paris it might be 5.120,00. The delimiters for thousands and the decimal—the comma and the period, respectively—are switched automatically according to the preferences of the locale.

Note: Changing the settings of the default locale can change the behavior of existing programs. For example, if you change the default locale to French, programs that previously printed dates in English can now print them in French. Be sure that you review and test existing programs when making a change to the default locale.

Personalizing a Default Locale

Starting with PeopleTools 8.54, users in different geographical regions can generate personalized SQR reports specific to the region. However, you can customize only the following default locale settings:

- DECIMAL-SEPARATOR
- THOUSAND-SEPARATOR
- DATE-SEPARATOR
- TIME-SEPARATOR
- DATE-EDIT-MASK

The default values for each locale setting will be used if personalization is enabled for SQR reports for a specific user. Personalization of SQR reports will be an option that the administrator will grant. By default, the feature remains off. To generate a region-specific report, the administrator must enable Personalize SQR Settings in the PIA. For more information about personalizing user, see "Understanding My Preferences Personalizations" (PeopleTools 8.58: Security Administration)..

Switching Locales

You can switch from one locale to another any number of times while a program runs. This technique is useful for writing reports that use multiple currencies, or reports that have different sections for different locales.

To switch to another locale, use the ALTER-LOCALE command. For example, to switch to the Spanish locale:

```
alter-locale locale = 'Spanish'
```

From this point in the program, the locale is Spanish.

Consider this code example:

```
begin-procedure print_data_in_spanish
   ! Save the current locale
   let $old_locale = $sqr-locale
   ! Change the locale to "Spanish"
   alter-locale locale = 'Spanish'
   ! Print the data
   do print_data
   ! restore the locale to the previous setting
   alter-locale locale = $old_locale
end-procedure
```

In this code example, the locale is switched to Spanish and later restored to the previous locale before it was switched. To do that, the locale setting before it is changed is read in the*$sqr-locale* reserved variable and stored in *$old_locale.* The value of *$old_locale* is then used in the ALTER-LOCALE command at the end of the procedure.

Modifying Locale Preferences

With the ALTER-LOCALE command, you can modify individual preferences in a locale. The ALTER-LOCALE command affects only the current program. It does not modify the pssqr.ini file.

Here is a code example of how you can modify default preferences in a locale:

```
alter-locale
    date-edit-mask  = 'Mon-DD-YYYY'
    money-edit-mask = '$$,$$$,$$9.99'
```

To restore modified locale preferences to their defaults, select the modified locale again. For example, suppose that the locale was US-English and the date and money edit masks were modified by using the preceding code. The following code resets the changed date and money edit masks:

```
alter-locale locale = 'US-English'
```

Specifying NUMBER, MONEY, and DATE Keywords

The DISPLAY, MOVE, PRINT, and SHOW commands enable you to specify the **NUMBER** , **MONEY**, and **DATE** keywords in place of an explicit number or date edit mask. These keywords can be useful in two cases.

The first case is when you want to write programs that automatically adapt to the default locale. By using the **NUMBER**, **MONEY**, and **DATE** keywords, you instruct SQR to take these edit masks from the default locale settings.

The second case is when you want to specify number, money, and date formats once at the top of the program and use these formats throughout the report. In this case, you define these formats with an ALTER-LOCALE command at the top of the program. When you use the **NUMBER**, **MONEY**, and **DATE** keywords later in the program, they format number, money, and date outputs with the masks that you defined in the ALTER-LOCALE command.

Whether you set the locale in the pssqr.ini file or in the program, these keywords have the same effect. In the following code example, these keywords are used with the PRINT command to produce output for the US-English and French locales:

```
let #num_var = 123456
let #money_var = 123456
let $date_var = strtodate('20040520152000')
! set locale to US-English
alter-locale locale = 'US-English'
print 'US-English locale' (1,1)
print 'With NUMBER keyword ' (+1,1)
print #num_var (,22) NUMBER
print 'With MONEY keyword ' (+1,1)
print #money_var (,22) MONEY
print 'With DATE keyword ' (+1,1)
print $date_var (,22) DATE! set locale to French
ALTER-LOCALE locale = 'French'
print 'French locale' (+2,1)
print 'With NUMBER keyword ' (+1,1)
print #num_var (,22) NUMBER
print 'With MONEY keyword ' (+1,1)
print #money_var (,22) MONEY
print 'With DATE keyword ' (+1,1)
print $date_var (,22) DATE
```

Here is the program output:

```
US-English locale
With NUMBER keyword  123,456.00
With MONEY keyword   $   123,456.00
With DATE keyword    May 20, 2004

French locale
With NUMBER keyword  123.456,00
With MONEY keyword   123.456,00 F
With DATE keyword    20 Mai 2004
```

Related Links

SQR Language Reference for PeopleSoft

Chapter 23

Using Interoperability Features

Calling SQR from Another Application

Applications can run SQR programs by using the SQR application program interface (API). An SQR program can also call the API of an external application.

To invoke an SQR program from another application, use:

- The SQR command line.

  The application initiates a process for running SQR. The SQR command includes all of the necessary parameters.

- The SQR API.

  The application makes a call to the SQR API. This method is covered in the next section.

See Understanding the SQR Command Line and Compiling SQR Programs and Using SQR Execute.

Invoking an SQR Program by Using the SQR API

The SQR API is provided in Microsoft Windows through a Dynamic Link Library (dll). You can use the SQR API from any application that is capable of calling dll functions. For C and C++ applications, a header file (sqrapi.h) and an import library (sqrwin.lib) are provided. SQR requires the following .dll files to run for Microsoft Windows: sqrw.dll, bclw32.dll, libsti32.dll, and stimages.dll. These dll files are located in the BINW directory.

On platforms other than Microsoft Windows, the SQR API is provided as a static library (sqr.a or sqr.lib). For C and C++ applications, a header file (SQRAPI.H or sqrapi.h) is provided. Be sure to include the SQR API library and your database library when you link your C or C++ application. Three additional libraries are required: sqrlibsti64.a , sqrbcl.a and sqrzlib.a.

The following table describes the API functions that are defined for calling SQR:

Function	***Description***
int sqr(char *)	Runs an SQR program. Passes the address of a null terminated string containing an SQR command line, including program name, connectivity information, flags, and arguments. This function is a synchronous call. It returns when the SQR program has finished. This function returns zero if it is successful.

Function	Description
void sqrcancel(void)	Cancels a running SQR program. The program may not stop immediately because SQR waits for any currently pending database operations to finish. Because the SQR function does not return until the SQR program has finished, sqrcancel is called by using another thread or some similar asynchronous method.
int sqrend(void)	Releases memory and closes cursors. Cursors can be left open to speed up repeated running of the same SQR program. Call this function after the last program has run or optionally between SQR program runs. This function always returns zero.

For the benefit of C and C++ programmers, the APIs are declared in the sqrapi.h file. Include this header file in your source code:

```
#include 'sqrapi.h'
```

When you call SQR from a program, the most recently run SQR program is saved in memory. If the same SQR program is run again with either the same or different arguments, the program is not scanned again and the SQL statements are not parsed again. This feature provides a significant improvement in processing time.

To force SQR to release its memory and database cursors, call *sqrend()* at any time.

Although memory is automatically released when the program exits, you must call *sqrend* before the calling program quits to ensure that SQR properly cleans up any database resources, such as database cursors and temporarily stored procedures.

To relink SQR on all UNIX/Linux platforms, use the sqrmake and makefile files that are located in $SQRDIR/../lib. After you invoke sqrmake and optionally select the specific database version to link with, the SQR executables are re-created.

Check which `cc` command line is created and invoked for SQR, and adapt it to your program. Each UNIX/Linux platform and database has its own requirements. Consult your operating system and database product documentation for specific information.

Check the make files or link scripts that are supplied with SQR for details. You may want to copy and modify those to link in your program.

To call SQR, call *sqr()* and pass a command line. For example, in C:

```
status = sqr("myprog sammy/baker arg1 arg2 arg3");
if (status != 0)
      ...error occurred...
```

The following table describes the standalone and callable error values that SQR returns:

Error Value	Description
0	Normal exit.

Error Value	*Description*
1	Error exit.
2	Cannot process SQRERR.DAT.
3	Command-line flag in error.
4	Problem creating the .SQT file.
5	Program did not compile.
6	Problem with the .SQR/.SQT file (open/read).
7	Problem with the .LIS file (create/write).
8	Problem with the .ERR file (create/write).
9	Problem with the .LOG file (create/write).
10	Problem with the POSTSCRI.STR file (open/read).
11	Cannot call SQR recursively.
12	Problem with Microsoft Windows.
13	Internal error occurred.
14	Problem with SQRWIN.DLL.
15	Problem with -ZCF file.

Error codes 9 and 12 are applicable to the Microsoft Windows release only.

For more information about linking with SQR, see your installation guide.

See the product documentation for *PeopleSoft 9.2 Application Installation*.

Invoking an External Application API by Using the UFUNC.C Interface

You can extend the SQR language by adding user functions that are written in standard languages, such as C. This feature enables you to integrate your own code and third-party libraries into SQR. For example, suppose that you have a library for communication over a serial line, with functions for initiating the connection and sending and receiving data. SQR enables you to call these functions from SQR programs.

To extend SQR in this way, you must prepare the functions, specify them to SQR, and then link the objects (and libraries) with the SQR objects and libraries to form a new SQR executable. The new SQR executable then recognizes the new functions as if they were standard SQR functions.

One example of such an extension would be an initcap function. Oracle users are familiar with this function. The initcap function changes the first letter of every word to uppercase and changes the rest of the letters to lowercase. The result value in the following code example would be *Mr. Joseph Jefferson:*

```
let $a = initcap('MR. JOSEPH JEFFERSON')
```

Adding a User Function

This section provides an overview of the ufunc.c file and discusses how to:

- Add a function prototype.
- Add an entry to the USERFUNCS table.
- Add implementation code.
- Relink SQR.

Understanding the UFUNC.C File

The code examples in the following sections demonstrate how to extend SQR with an initcap function.

The key to this process is an SQR source file called ufunc.c. This file contains a list of user-defined functions. It also contains comments with a description of the process of adding a function to SQR. Ufunc.c is in the lib subdirectory (LIBW in Microsoft Windows).

To add initcap to SQR, you must add it to a global array called userfuncs in ufunc.c.

Adding a Function Prototype

Begin by adding a function prototype to the function declaration list:

```
static void max CC_ARGS((int, double *[], double *));
static void split CC_ARGS((int, char *[], double *));
static void printarray CC_ARGS((int, char*[], double *));
static void initcap CC_ARGS((int, char *[], char *, int));
```

The preceding code segment is taken from the ufunc.c file. The first three lines are part of the original ufunc.c file. The line that adds the initcap function is shown **like this**. The modified version of ufunc.c is in the LIBW (Microsoft Windows) or LIB (UNIX) directory under <PS_HOME>\bin\sqr \<database_platform>.

This code defines a prototype for a C function called initcap. The prototype is required by the C compiler. The name of the C function does not have to be the same as the name of the SQR function. The SQR name for the function is defined in the next step.

The CC_ARGS macro makes the code portable between compilers that expect full prototyping and compilers in which the argument prototype is omitted. You could also write:

```
static void initcap();
```

Note also that the STATIC keyword means that the code for initcap will be added in the file ufunc.c. If you have the code in a separate file, remove the STATIC keyword.

The first argument of the C function is the argument count of the corresponding SQR function. In the case of initcap, this argument count should be 1 because initcap takes exactly one argument.

The second argument of the C function is an array of pointers. This array is the argument list. In this case, because initcap takes only one argument, only the first pointer is actually used.

The third argument of the C function is a pointer to the result buffer. Because initcap returns a string, it is defined as char*.

The last argument sets the maximum length of the result string. The length of this string is the size of the result buffer, which you must not overflow. You cannot return a value that is longer than the maximum length. The maximum length is typically around 2000 bytes, depending on the platform.

Adding an Entry to the USERFUNCS Table

The next step is to define the initcap function to SQR. As stated before, this table exists in the ufunc.c file. Here is the modified code:

```
} userfuncs[] =
 {
 /* (2) Define functions in userfuncs table:
                           Number of
 Name          Return_type Arguments Arg_Types Function
 ----          ----------- --------- --------- -------- */
 "max",        'n',        0,        "n",      PVR max,
 "split",      'n',        0,        "C",      PVR split,
 "printarray", 'n',        4,        "cnnc",   PVR printarray,
 "initcap",    'c',        1,        "c",      PVR initcap,
 /* Last entry must be NULL do not change */
 "", '\0', 0, "", 0
};
```

The userfuncs table is an array of structures. The added line is shown **like this,** and it initializes one structure in the array. The line contains five arguments, which correspond to the five fields of the structure.

The first argument is the name of the SQR function that is being added. This is the name that you will use in the LET, IF, and WHILE commands. The second argument is the return type, which 'c' (enclosed in single quotation marks) indicates is a character string. The third argument is the number of arguments that initcap will take. Set it to 1.

The fourth argument is a string representing the types of the arguments. Because initcap has only one argument, the string contains one character enclosed in double quotation marks, "c". This character indicates that the argument for initcap is a string. The last argument is a pointer to a C function that implements the SQR function that you are adding. This argument is the initcap function for which we have provided a prototype in the previous step. Note that the PVR macro provides proper cast for the pointer.

Adding Implementation Code

The next step is to add the implementation code for initcap. You can insert it into the ufunc.c file.

Note: To put the code in a separate file, you must remove the STATIC keyword from the prototype. You may also need to include standard C header files, such as CTYPE.H.

Here is the code that is inserted at the end of ufunc.c:

```
static void initcap CC_ARGL((argc,argv,result,maxlen))
CC_ARG(int, argc)       /* Number of actual arguments */
CC_ARG(char*, argv[])   /* Pointers to arguments: */
CC_ARG(char*, result)   /* Where to store result */
CC_LARG(int, maxlen)    /* Result's maximum length */
{
  int flag = 1;
  char *ptr;
  char *p;
  ptr = argv[0];
  p = result;
  while (*ptr) {
    if (ptr - argv[0] >= maxlen) break; /* don't exceed maxlen */
    if (isalnum(*ptr)) {
       if (flag) *p = islower(*ptr)?toupper(*ptr):*ptr;
       else *p = isupper(*ptr)?tolower(*ptr):*ptr;
       flag = 0;
    } else {
       flag = 1;
       *p = *ptr;
    }
     p++; ptr++;
  }
  *p = '\0';
  return;
}
```

Note the use of the CC_ARGL, CC_ARG, and CC_LARG macros. You can also write the code as follows (only the first five lines are shown):

```
static void initcap(argc,argv,result,maxlen)
int argc;       /* Number of actual arguments */
char* argv[];   /* Pointers to arguments: */
char* result;   /* Where to store result */
int maxlen;     /* Result's maximum length */
```

Relinking SQR

After you modify ask, you must relink SQR. Use the make file that is provided in the LIB (or LIBW) subdirectory of SQR. This step is specific to the operating system and database. SQR is linked with the database libraries, whose names and locations may vary. You may have to modify the make file for your system.

After SQR is relinked, you are ready to test. Try the following program:

```
begin-program
  let $a = initcap('MR. JOSEPH JEFFERSON')
  print $a ()
end-program
```

The result in the output file should be:

```
Mr. Joseph Jefferson
```

See the ufunc.c file for further information about argument types in user-defined functions.

See the product documentation for *PeopleSoft 9.2 Application Installation*.

Using UFUNC in Microsoft Windows

In Microsoft Windows, ufunc resides in sqrext.dll. You can rebuild sqrext.dll by using any language or tool, as long as you maintain the appropriate calling protocol. The source code for sqrext.dll is included in the shipped package (extufunc.c).

When sqrw.dll and sqrwt.dll are loaded, they look for sqrext.dll in the same directory and for any .dlls that are specified in the SQR Extension section in pssqr.ini. If sqrw.dll and sqrwt.dll find sqrext.dll and the .dlls that are specified in the pssqr.ini file, they make the following calls in all of the .dlls, passing the instance handle (of the calling module) and three function pointers:

```
void InitSQRExtension (
   HINSTANCE hInstance,
   FARPROC lpfnUFuncRegister,
   FARPROC lpfnConsole,
   FARPROC lpfnError
   );
```

Implementing New User Functions in Microsoft Windows

You can implement new user functions in sqrext.dll or any other extension .dll. All of the extension .dlls must have the InitSQRExtension() function exported. If you implement user functions in sqrext.dll, you should rebuild the .dll by using the supplied make file, sqrext.mak. If new extension .dlls containing new user functions are to be used, they must be listed in the SQR Extension section in pssqr.ini in the system directory.

For any ufunc, you must register it by making the following call in InitSQRExtension():

```
lpfnUFuncRegister(struct ufnns* ufunc);
```

The function pointer, **lpfnUFuncRegister**, is passed in from the calling module. Refer to extufunc.c for the definition of *struct ufnns* and the sample user functions.

Chapter 24

Testing and Debugging

Using the Test Feature

When developing an SQR program, you frequently test it by running it and examining its output. Often, you are interested only in the first few pages of a report.

To speed up the cycle of running and viewing a few pages, use the -T command-line flag. The -T flag enables reports to finish more quickly because all BEGIN-SELECT ORDER BY clauses are ignored. The database does not sort the data, and the first set of records is selected sooner. Enter the appropriate number of test pages following the -T flag. For example, -T6 causes the program to stop after six pages of output are created.

Note: If the program contains break logic, the breaks can occur in unexpected locations because the ORDER BY clause is ignored.

To test a report file called customer.sqr, enter the following command:

```
sqr customer username/password -T3
```

The -T3 flag specifies that the program stops running after three pages are produced.

When the test finishes successfully, check it by displaying the output file on the screen or by printing it. The default name of the output file is the same as the program file with the .LIS extension. For example, if the report is named customer.sqr, then the output file is named customer.lis.

When you finish developing the program, run it without the -T flag. The program processes all ORDER BY clauses and runs to completion. If the program creates more than one report, the -T flag restriction applies only to the first report.

Using the #DEBUG Command

When debugging a program, you should:

- Display data or show when a procedure or query runs by using temporary SHOW or DISPLAY commands in key places in the program.
- Isolate problem areas by temporarily skipping the parts of the program that work correctly.
- Temporarily cause additional behavior in questionable areas of the program.

  For example, display or modify variables that you suspect are causing a problem.

SQR provides the #DEBUG command to help you make temporary changes to the code. Use the #DEBUG command to conditionally process portions of the program.

Precede the command with #DEBUG, as shown in the following example:

```
#debug display $s
```

When the #DEBUG precedes a command, that command is processed only if the -DEBUG flag is specified in the SQR command line. In this example, the value of *$s* appears only when you run the program with -DEBUG.

You can obtain multiple debug commands by using up to 10 letters or digits to differentiate between them. Indicate which command is to be debugged on the -DEBUG flag, as shown in the following example:

```
sqr myreport username/password -DEBUGabc
```

In this example, commands that are preceded by #DEBUG, #DEBUGa, #DEBUGb, or #DEBUGc are compiled when the program is run. Commands that are preceded with #DEBUGd are not compiled because *d* was not specified in the -DEBUG command-line flag.

Using Compiler Directives for Debugging

You can conditionally compile entire sections of a program by using the five compiler directives:

- #IF
- #ELSE
- #END-IF or #ENDIF
- #IFDEF
- #IFNDEF

Use the value of a substitution variable, declared by a #DEFINE command, to activate or deactivate a set of statements, as shown in the following example:

```
#define DEBUG_SESSION Y
#if DEBUG_SESSION = 'Y'
begin-procedure dump_array
   let #i = 0
   while #i < #counter
      ! Get data from the array
      get $state $city $name $phone from customer_array(#i)
      print $state (,1)
      print $city  (,7)
      print $name  (,24)
      print $phone (,55)
      position (+1)
      add 1 to #i
   end-while
end-procedure ! dump_array
#end-if
```

The dump_array procedure is used only for debugging. Because DEBUG_SESSION is defined as Y, the dump_array procedure is included in the program. Later, you can change DEBUG_SESSION to N and exclude the dump_array procedure from the program.

Avoiding Common Programming Errors

The most common programming error when you are using SQR is misspelling variable names. Because SQR does not require variables to be declared, it does not issue an error message when variable names are misspelled. Instead, SQR considers the misspelled variable as if it is another variable.

For example:

```
let #customer_access_code = 55
print #customer_acess_code ()
```

This example does not print 55 because the variable name is misspelled. One *c* in *access* in the PRINT command is missing.

A related problem involves global versus local variables. If you refer to a global variable in a local procedure without preceding it with an underscore, SQR does not issue an error message. Instead, it is taken as a new local variable name. For example:

```
begin-procedure main
   let $area = 'North'
   do proc
end-procedure ! main
begin-procedure proc local
   print $area ()  ! Should be $_area
end-procedure
```

In this example, the **proc local** procedure prints the value of the local *$area* variable and not the global *$area* variable. It prints nothing because the local *$area* variable did not receive a value. To refer to the global variable, use *$_area.*

Such small errors are difficult to detect because SQR considers *#customer_acess_code* as just another variable with a value of zero.

Chapter 25

Increasing Performance and Tuning

Understanding SQR Performance and SQL Statements

Whenever a program contains a BEGIN-SELECT, BEGIN-SQL, or EXECUTE command, it performs a SQL statement. Processing SQL statements typically consumes significant computing resources. Tuning SQL statements typically yields higher performance gains than tuning any other part of the program.

General tuning of SQL is outside the scope of this PeopleBook because tuning SQL is often specific to the type of database that you are using: tuning SQL statements for an Oracle database may be different from tuning SQL statements for DB2. This topic focuses on SQR tools for simplifying SQL statements and reducing the number of times SQL is run.

Simplifying Complex Select Paragraphs

With relational database design, information is often normalized by storing data entities in separate tables. To display the normalized information, you must write a select paragraph that joins these tables. With many database systems, performance suffers when you join more than three or four tables using one select paragraph.

With SQR, you can perform multiple select paragraphs and nest them. In this way, you can break a large join into several simpler selects. For example, you can break a select paragraph that joins the orders and the products tables into two selects. The first select retrieves the orders that you want. For each order that is retrieved, a second select retrieves the products that were ordered. The second select is correlated to the first select by a condition such as:

```
where order_num = &order_num
```

This condition specifies that the second select retrieves only products for the current order.

Similarly, if a report is based on products that were ordered, you can make the first select retrieve the products and the second select retrieve the orders for each product.

This method improves performance in many cases, but not all. To achieve the best performance, you may need to experiment with the different alternatives.

You can use master and detail reports to perform multiple select paragraphs and nest them.

See Using Dynamic SQL.

Using LOAD-LOOKUP to Simplify Joins

Database tables often contain key columns, such as a product code or customer number. To retrieve a certain piece of information, you join two or more tables that contain the same column. For example, to obtain a product description, you can join the orderlines table with the products table by using the product_code column as the key.

With LOAD-LOOKUP, you can reduce the number of tables that are joined in one select. Use this command with LOOKUP commands.

The LOAD-LOOKUP command defines an array containing a set of keys and values, and loads it into memory. The LOOKUP command looks up a key in the array and returns the associated value. In some programs, this technique performs better than a conventional table join.

You can use LOAD-LOOKUP in the SETUP section or in a procedure. If used in the SETUP section, it is processed only once. If used in a procedure, it is processed each time that it is encountered.

LOAD-LOOKUP retrieves two fields from the database: the KEY field and the RETURN_VALUE field. Rows are ordered by KEY and stored in an array. The KEY field must be unique and contain no null values.

When the LOOKUP command is used, the array is searched (using a binary search) to find the RETURN_VALUE field corresponding to the KEY that is referenced in the lookup.

The following code example illustrates LOAD-LOOKUP and LOOKUP:

```
begin-setup
  load-lookup
     name=prods
      table=products
      key=product_code
      return_value=description
end-setup
...
begin-select
order_num (+1,1)
product_code
  lookup prods &product_code $desc
  print $desc (,15)
from orderlines
end-select
```

In this code example, the LOAD-LOOKUP command loads an array with the product_code and description columns from the products table. The lookup array is named prods. The product_code column is the key, and the description column is the return value. In the select paragraph, a LOOKUP on the prods array retrieves the description for each product_code. This technique eliminates the need to join the products table in the select.

If the orderlines and products tables were joined in the select (without LOAD-LOOKUP), the code would look like this:

```
begin-select
order_num (+1,1)
ordlines.product_code
description (,15)
from ordlines, products
where ordlines.product_code = products.product_code
end-select
```


Whether a database join or LOAD-LOOKUP is faster depends on the program. LOAD-LOOKUP improves performance when:

- It is used with multiple select paragraphs.
- It keeps the number of tables being joined from exceeding three or four.
- The number of entries in the LOAD-LOOKUP table is small compared with the number of rows in the select, and they are used often.
- Most entries in the LOAD-LOOKUP table are used.

Note: You can concatenate columns if you want RETURN_VALUE to return more than one column. The concatenation symbol is database specific.

Improving SQL Performance with Dynamic SQL

You can use dynamic SQL in some situations to simplify a SQL statement and gain performance:

```
begin-select
order_num
from orders, customers
where order.customer_num = customers.customer_num
and ($state = 'CA' and order_date > $start_date
     or $state != 'CA' and ship_date > $start_date)
end-select
```

In this example, a given value of `$state, order_date,` or `ship_date` is compared with `$start_date.` The OR operator in the condition makes such multiple comparisons possible. With most databases, an OR operator slows processing. It can cause the database to perform more work than necessary.

However, the same work can be done with a simpler select. For example, if `$state` is ‘CA,’ then the following select works:

```
begin-select
order_num
from orders, customers
where order.customer_num = customers.customer_num
and order_date > $start_date
end-select
```

Dynamic SQL enables you to check the value of `$state` and create the simpler condition:

```
if $state = 'CA'
   let $datecol = 'order_date'
else
   let $datecol = 'ship_date'
end-if
begin-select
order_num
from orders, customers
where order.customer_num = customers.customer_num
and [$datecol] > $start_date
end-select
```


The *[$datecol]* substitution variable substitutes the name of the column to be compared with `$start_date`. The select is simpler and no longer uses an OR operator. In most cases, this use of dynamic SQL improves performance.

See Using Dynamic SQL.

Examining SQL Cursor Status

Because SQR programs select and manipulate data from a SQL database, you should understand how SQR processes SQL statements and queries.

SQR programs can perform multiple SQL statements. Moreover, they can run the same SQL statement multiple times.

When a program runs, a pool of SQL statement handles, called cursors, is maintained. A cursor is a storage location for one SQL statement—for example, SELECT, INSERT, or UPDATE. Every SQL statement uses a cursor for processing. A cursor holds the context for the execution of a SQL statement.

The cursor pool contains 30 cursors, and you cannot change its size. When a SQL statement is rerun, its cursor can be immediately reused if it is still in the cursor pool. When an SQR program runs more than 30 different SQL statements, cursors in the pool are reassigned.

To examine how cursors are managed, use the -S command-line flag. This flag displays cursor status information at the end of a run.

The following information appears for each cursor:

```
Cursor #nn:
SQL = <SQL statement>
Compiles = nn
Executes = nn
Rows = nn
```

The listing also includes the number of compiles, which varies according to the database and the complexity of the query. With Oracle, for example, a simple query is compiled only once, while on other database platforms, a SQL statement may be compiled before it is first run and recompiled for the purpose of validation during the SQR compile phase. Therefore, you may see two compiles for a SQL statement. Later, when the SQL is rerun, if its cursor is found in the cursor pool, then it can proceed without recompiling.

Avoiding Temporary Database Tables

This section provides an overview of temporary database tables and discusses how to:

- Use and sort arrays.
- Use and sort flat files.

Understanding Temporary Database Tables

Programs often use temporary database tables to hold intermediate results. Creating, updating, and deleting temporary tables is a resource consuming task, however, and can slow program performance. SQR provides two alternatives to using temporary database tables:

- Store intermediate results in an SQR array.
- Store intermediate results in a local flat file.

Both techniques can yield a significant performance gain. Use the SQR language to manipulate the data that is stored in an array or a flat file.

Using and Sorting Arrays

An SQR array can hold as many records as can fit in memory. During the first pass, when records are retrieved from the database, you can store them in the array. Subsequent passes on the data can be made without additional database access.

The following code example retrieves records, prints them, and saves them to an array named customer_array:

```
create-array name=customer_array size=1000
   field=state:char   field=city:char
   field=name:char    field=phone:char
let #counter = 0
begin-select
state (,1)
city  (,7)
name  (,24)
phone (,55)
   position (+1)
   put &state &city &name &phone into customer_array(#counter)
   add 1 to #counter
from customers
end-select
```

The customer_array array has four fields that correspond to the four columns selected from the customer's table, and it can hold up to 1,000 rows. If the customer's table had more than 1,000 rows, you would need to create a larger array.

The select paragraph prints the data. The PUT command then stores the data in the array. You could use the LET command to assign values to array fields; however, the PUT command performs the same work with fewer lines of code. With PUT, you can assign all four fields in one command.

The *#counter* variable serves as the array subscript. It starts with zero and maintains the subscript of the next available entry. At the end of the select paragraph, the value of *#counter* is the number of records in the array.

The next code example retrieves the data from customer_array and prints it:

```
let #i = 0
while #i < #counter
   get $state $city $name $phone from customer_array(#i)
   print $state (,1)
   print $city  (,7)
   print $name  (,24)
   print $phone (,55)
   position (+1)
   add 1 to #i
```



```
end-while
```

In this code example, *#i* goes from 0 to *#counter*–1. The fields from each record are moved into the corresponding variables: *$name, $city, $state,* and *$phone.* These values are then printed.

Sorting Arrays

In many cases, intermediate results must be sorted by a different field. The following sample program shows how to sort customer_array by name. The sample program uses a well-known sorting algorithm called QuickSort. You can copy this code into your program, make appropriate changes, and use it to sort your array:

```
Program ex24a.sqr
#define MAX_ROWS 1000
begin-setup
create-array name=customer_array size={MAX_ROWS}
   field=state:char     field=city:char
   field=name:char      field=phone:char
!
! Create a helper array that is used in the sort
!
create-array name=QSort size={MAX_ROWS}
      field=n:number    field=j:number
end-setup
begin-program
  do main
end-program
begin-procedure main
let #counter = 0
!
! Print customers sorted by state
!
begin-select
state (,1)
city  (,7)
name  (,24)
phone (,55)
   position (+1)
   ! Put data in the array
   put &state &city &name &phone into customer_array(#counter)
   add 1 to #counter
from customers
order by state
end-select
position (+2)
!
! Sort customer_array by name
!
let #last_row = #counter - 1
do QuickSort(0, 0, #last_row)
!
! Print customers (which are now sorted by name)
!
let #i = 0
while #i < #counter
   ! Get data from the array
   get $state $city $name $phone from customer_array(#i)
   print $state (,1)
   print $city  (,7)
   print $name  (,24)
   print $phone (,55)
   position (+1)
   add 1 to #i
end-while
end-procedure ! main
!
! QuickSort
!
```



```
! Purpose: Sort customer_array by name.
! This is a recursive function. Since SQR does not allocate
! local variables on a stack (they are all static), this
! procedure uses a helper array.
!
! #level - Recursion level (used as a subscript to the helper
! array)
! #m     - The "m" argument of the classical QuickSort
! #n     - The "n" argument of the classical QuickSort
!
begin-procedure QuickSort(#level, #m, #n)
   if #m < #n
      let #i = #m
      let #j = #n + 1
      ! Sort key is "name"
      let $key = customer_array.name(#m)
      while 1
         add 1 to #i
         while #i <= #j and customer_array.name(#i) < $key
             add 1 to #i
         end-while
         subtract 1 from #j
         while #j >= 0 and customer_array.name(#j) > $key
            subtract 1 from #j
         end-while
         if #i < #j
            do QSortSwap(#i, #j)
         else
            break
         end-if
      end-while
      do QSortSwap(#m, #j)
      add 1 to #level
      ! Save #j and #n
      let QSort.j(#level - 1) = #j
      let QSort.n(#level - 1) = #n
      subtract 1 from #j
      do QuickSort(#level, #m, #j)
      ! restore #j and #n
      let #j = QSort.j(#level - 1)
      let #n = QSort.n(#level - 1)
      add 1 to #j
      do QuickSort(#level, #j, #n)
      subtract 1 from #level
   end-if
end-procedure ! QuickSort
!
!
! QSortSwap
!
! Purpose: Swaps records #i and #j of customer_array
 !
! #i     - Array subscript
! #j     - Array subscript
!
begin-procedure QSortSwap(#i, #j)
   get $state $city $name $phone from customer_array(#i)
   let customer_array.state(#i) = customer_array.state(#j)
   let customer_array.city(#i)  = customer_array.city(#j)
   let customer_array.name(#i)  = customer_array.name(#j)
   let customer_array.phone(#i) = customer_array.phone(#j)
   put $state $city $name $phone into customer_array(#j)
end-procedure ! QSortSwap
```

The QuickSort algorithm uses a recursive procedure, which means that it calls itself. SQR maintains only one copy of the local variables of the procedure. In QuickSort, the *#j* and *#n* variables are overwritten when QuickSort calls itself.

For the algorithm to work properly, the program must save the values of these two variables before making the recursive call, and then it must restore those values when the call finishes. QuickSort can call itself recursively many times, so the program may need to save many copies of *#j* and *#n*. To have the program do this, add a *#level* variable that maintains the depth of recursion. In this example, a helper array, Qsort, is used to hold multiple values of *#j* and *#n*.

The QuickSort procedure takes three arguments. The first is the recursion level (or depth), which is *#level,* as previously described. The second and third arguments are the beginning and end of the range of rows to be sorted. Each time QuickSort calls itself, the range becomes smaller. The main procedure starts QuickSort by calling it with the full range of rows.

The QSortSwap procedure swaps two rows in customer_array. Typically, rows with a lower key value are moved up.

The QuickSort and QSortSwap procedures in ex24a.sqr refer to customer_array and its fields. If you plan to use these procedures to sort an array in your applications, you must change these references to the applicable array and fields. The QuickSort procedure sorts in ascending order.

SQR and Language-Sensitive Sorting

SQR does not support National Language Sensitive sorting natively. SQR compares characters based on Unicode codepoint, and sorting based on Unicode codepoint does not correctly sort order language-sensitive data.

See Understanding the SQR Command Line.

The QuickSort procedure does not support National Language Sensitive character string sorting. The comparisons are simple string comparisons based on Unicode codepoint used internally in SQR to represent string data. For instance, the following code lines from the preceding code sample would sort data in Unicode codepoint order. Unicode codepoints are not ordered to make a correct sorting order of any language.

```
while #i <= #j and customer_array.name(#i) < $key
and
while #j >= 0 and customer_array.name(#j) > $key
```

If you want to sort string data in SQR, you may need to write a National Language Sensitive character string comparison and add that to SQR. The QuickSort procedure will then be modified in the following way:

```
while #i <= #j and NLS_STRING_COMPARE(customer_array.name(#i),$key)
while #j >= 0 and NLS_STRING_COMPARE($key,customer_array.name(#j))
```

Using and Sorting Flat Files

An alternative to an array is a flat file. You can use a flat file when the required array size exceeds the available memory.

The code example in the previous section can be rewritten to use a file instead of an array. The advantage of using a file is that the program is not constrained by the amount of memory that is available. The disadvantage of using a file is that the program performs more input and output (I/O). However, it may still be faster than performing another SQL statement to retrieve the same data.

This program uses the UNIX/Linux sort utility to sort the file by name. This example can be extended to include other operating systems.

The following code example is rewritten to use the cust.dat file instead of the array:

```
Program ex24b.sqr
begin-program
  do main
end-program
begin-procedure main
!
! Open cust.dat
!
open 'cust.dat' as 1 for-writing record=80:vary
begin-select
state (,1)
city  (,7)
name  (,24)
phone (,55)
   position (+1)
   ! Put data in the file
   write 1 from &name:30 &state:2 &city:16 &phone:10
from customers
order by state
end-select
position (+2)
!
! Close cust.dat
close 1
! Sort cust.dat by name
!
call system using 'sort cust.dat > cust2.dat' #status
if #status <> 0
    display 'Error in sort'
    stop
end-if
!
! Print customers (which are now sorted by name)
!
open 'cust2.dat' as 1 for-reading record=80:vary
while 1  ! loop until break
   ! Get data from the file
   read 1 into $name:30 $state:2 $city:16 $phone:10
   if #end-file
      break   ! End of file reached
   end-if
   print $state (,1)
   print $city  (,7)
   print $name  (,24)
   print $phone (,55)
   position (+1)
end-while
!
! close cust2.dat
close 1
end-procedure ! main
```

The program starts by opening a cust.dat file:

```
open 'cust.dat' as 1 for-writing record=80:vary
```

The OPEN command opens the file for writing and assigns it file number 1. You can open as many as 12 files in one SQR program. The file is set to support records of varying lengths with a maximum of 80 bytes (characters). For this example, you can also use fixed-length records.

As the program selects records from the database and prints them, it writes them to cust.dat:

```
write 1 from &name:30 &state:2 &city:16 &phone:10
```

The WRITE command writes the four columns into file number 1, the currently open cust.dat. It writes the name first, which simplifies sorting the file by name. The program writes fixed-length fields. For

example, `&name:30` specifies that the name column uses exactly 30 characters. If the actual name is shorter, it is padded with blanks. When the program has finished writing data to the file, it closes the file by using the CLOSE command.

The file is sorted with the UNIX sort utility:

```
call system using 'sort cust.dat > cust2.dat' #status
```

The `sort cust.dat > cust2.dat` command is sent to the UNIX system. It invokes the UNIX sort command to sort cust.dat and direct the output to cust2.dat. The completion status is saved in `#status;` a status of 0 indicates success. Because name is at the beginning of each record, the file is sorted by name.

Next, open cust2.dat for reading. The following command reads one record from the file and places the first 30 characters in `$name:`

```
read 1 into $name:30 $state:2 $city:16 $phone:10
```

The next two characters are placed in `$state,` and so on. When the end of the file is encountered, the *#end-file* reserved variable is automatically set to 1 (true). The program checks for *#end-file* and breaks out of the loop when the end of the file is reached. Finally, the program closes the file by using the CLOSE command.

Creating Multiple Reports in One Pass

Sometimes you must create multiple reports that are based on the same data. In many cases, these reports are similar, with only a difference in layout or summary. Typically, you can create multiple programs and even reuse code. However, if each program is run separately, the database has to repeat the query. Such repeated processing is often unnecessary.

With SQR, one program can create multiple reports simultaneously. In this method, a single program creates multiple reports, making just one pass on the data and reducing the amount of database processing.

See Understanding the Sample Program for Multiple Reports.

Tuning SQR Numerics

SQR for PeopleSoft provides three types of numeric values:

- Machine floating point numbers
- Decimal numbers
- Integers

Machine floating point numbers are the default. They use the floating point arithmetic that is provided by the hardware. This method is very fast. It uses binary floating point and normally holds up to 15 digits of precision.

Some accuracy can be lost when you are converting decimal fractions to binary floating point numbers. To overcome this loss of accuracy, you can sometimes use the ROUND option of commands such as

ADD, SUBTRACT, MULTIPLY, and DIVIDE. You can also use the round function of LET or numeric edit masks that round the results to the needed precision.

Decimal numbers provide exact math and precision of up to 38 digits. Math is performed in the software. This is the most accurate method, but also the slowest.

You can use integers for numbers that are known to be integers. Using integers is beneficial because they:

- Enforce the integer type by not allowing fractions.
- Adhere to integer rules when dividing numbers.

Integer math is also the fastest method, typically faster than floating point numbers.

If you use the DECLARE-VARIABLE command, the -DNT command-line flag, or the DEFAULT-NUMERIC entry in the Default-Settings section of the PSSQR.INI file, you can select the type of numbers that SQR uses. Moreover, you can select the type for individual variables in the program with the DECLARE-VARIABLE command. When you select decimal numbers, you can also specify the needed precision.

Selecting the numeric type for variables enables you to fine-tune the precision of numbers in your program. For most applications, however, this type of tuning does not yield a significant performance improvement, so selecting decimal is best. The default is machine floating point to provide compatibility with older releases of the product.

Compiling SQR Programs and Using SQR Execute

Compiling an SQR program can improve its performance. The compiled program is stored in a runtime (.SQT) file. You can then run it with SQR Execute. Your program runs faster because it bypasses the compile phase.

See SQR for PeopleSoft Overview and Understanding the SQR Command Line.

Setting Processing Limits

Use a startup file and the Processing-Limits section of pssqr.ini to define the sizes and limitations of some of the internal structures that SQR uses. An -M command-line flag can specify a startup file whose entries override those in pssqr.ini. If you use the -Mb command-line flag, then corresponding sections of the file are not processed. Many of these settings have a direct effect on memory requirements.

Tuning of memory requirements used to be a factor with older, 16-bit operating systems, such as Microsoft Windows 3.1. Today, most operating systems use virtual memory, and tuning memory requirements normally do not affect performance in any significant way. The only case in which you might need to be concerned with processing limit settings is with large SQR programs that exceed default processing limit settings. In such cases you must increase the corresponding settings.

Buffering Fetched Rows

When you run a BEGIN-SELECT command, SQR fetches records from the database server. For better performance, SQR fetches them in groups rather than one at a time—by default in groups of 10 records. SQR buffers the records, and a program processes these records one at a time. SQR, therefore, performs a database fetch operation after every 10 records instead of after every single record, which is a substantial performance gain. If the database server is on another computer, network traffic is also significantly reduced.

Modify the number of records to fetch together by using the -B command-line flag or, for an individual BEGIN-SELECT command, by using its -B option. In both cases, specify the number of records to be fetched together. For example, -B100 specifies that records be fetched in groups of 100. This means that the number of database fetch operations is further reduced.

This feature is currently available with SQR for Oracle database and SQR for ODBC.

Running Programs on the Database Server

To reduce network traffic and improve performance, run SQR programs directly on the database server machine. The SQR server is available on many server platforms, including Microsoft Windows and UNIX/Linux.

Chapter 26

Compiling Programs and Using SQR Execute

Understanding Compile Features

The following table lists SQR features that apply at compile time and their possible runtime equivalents. In some cases, no equivalent exists and you must work around the limitation. For example, you may have to use substitution variables with commands that require a constant and do not allow a variable. The topic "Writing Printer-Independent Reports" includes an example that works around the limitation of the USE-PRINTER-TYPE command, which does not accept a variable as an argument.

See Understanding the Sample Program for Multiple Reports and Understanding Compile Features.

Compile Time	***Runtime***
Substitution variables	Use regular SQR variables. If you are substituting parts of a SQL statement, use dynamic SQL instead. See Improving SQL Performance with Dynamic SQL
ASK	INPUT
#DEFINE	LET
#IF	IF
INCLUDE	No equivalent
DECLARE-LAYOUT, margins	No equivalent
Number of heading or footing lines	No equivalent
DECLARE-CHART	PRINT-CHART
DECLARE-IMAGE	PRINT-IMAGE
DECLARE-PROCEDURE	USE-PROCEDURE
DECLARE-PRINTER	ALTER-PRINTER (where possible)

Compiling and Running an SQR Program

For users, running an SQR program is a one-step process. For SQR, however, two steps are involved: compiling the program and running it. When compiling a program, SQR:

- Reads, interprets, and validates the program.
- Preprocesses substitution variables and certain commands: ASK, #DEFINE, #INCLUDE, #IF, and #IFDEF.
- Validates SQL statements.
- Performs the SETUP section.

Note: Make sure that SQRBIN (defined in psprcs.cfg) points to the correct location (PS_HOME/bin/SQR/<DB>/bin for Unix and PS_HOME/bin/sqrw/<DB>/BINW for Microsoft Windows) before you run an SQR program.

SQR enables you to save the compiled version of a program and use it when you rerun a report. That way, you perform the compile step only once and bypass it in subsequent runs. SQR does not compile the program into machine language. SQR creates a ready-to-run version of the program that is already compiled and validated. This file is portable between different hardware platforms and between some databases.

Run the SQR executable (SQR for UNIX/Linux or SQRW for Microsoft Windows) against the SQR program file and include the -RS command-line flag to save the runtime file. SQR creates a file with a file name extension of .sqt . You should enter something like this:

```
sqrw ex1a.sqr sammy/baker@rome -RS
```

Run the SQR executable with the -RT command-line flag to run the .sqt file. It runs faster because the program is already compiled. Here is an example:

```
sqrw ex1a.sqt sammy/baker@rome -RT
```

The SQR product distribution includes SQR Execute (the SQRT program). SQR Execute can run .sqt files, but it does not include the code that compiles an SQR program. (This program is equivalent to running SQR with -RT.) Here is an example of running SQR Execute from the command line:

```
sqrwt ex1a.sqt sammy/baker@rome
```

After you save the runtime (.sqt) file, SQR no longer performs any compile-time steps such as running #IF, #INCLUDE, or ASK commands or performing the SETUP section. These were already performed when the program was compiled and the runtime file was saved.

You must clearly distinguish between what action is performed at compile time and what action is performed at runtime. Think of compile-time steps as defining what a report is. Commands such as #IF or ASK enable you to adapt your report at compile time. For runtime adaptation, use commands such as IF and INPUT.

Chapter 27

Printing with SQR

Specifying Output File Types by Using SQR Command-Line Flags

Except on the Microsoft Windows platform, SQR does not actually print a report. SQR creates an output file that contains the report, but it does not print it directly. The output file can be a printer-specific file or an SQR portable file (SPF). SQR portable files have a default extension of .spf or .snn (for multiple reports).

The following table summarizes SQR command-line flags and the types of output that they produce:

Command-Line Flag	*Output File Extension*	*File Format*	*Suitable Usage*
-PRINTER:EH	.htm	Enhanced HTML	Intranet or internet
-PRINTER:HP	.lis	PCL	HP LaserJet printer
-PRINTER:HT	.htm	HTML	Intranet and internet
-PRINTER:LP	.lis	US ASCII	Line printer
-PRINTER:PS	.lis	PostScript	PostScript printer
-PRINTER:WP	None. Output goes directly to the default printer without being saved to a file. You can set the default printer by using the Microsoft Windows Control Panel.	Not applicable	Microsoft Windows
-NOLIS	.spf or .snn	SQR Portable file	SQR Print and SQR Viewer can print this file to different printers.
-KEEP	.spf or .snn (in addition to the .lis file that is normally created)	SQR Portable file and the format of the .lis file	SQR Print and SQR Viewer can print this .spf file to different printers.
No flag	.lis	US ASCII, PCL, or PostScript	Line printer, HP LaserJet, or PostScript, respectively

Note: When no flags are specified, SQR produces a line printer output unless it is otherwise set in the SQR program with DECLARE-PRINTER, USE-PRINTER-TYPE, or the PRINTER-TYPE option of DECLARE-REPORT.

SPF is a printer-independent file format that supports all of the SQR graphical features, including fonts, lines, boxes, shaded areas, charts, bar codes, and images.

This file format is useful for saving the output of a report. SPFs can be distributed electronically and read with the SQR Viewer. Producing SPF output also enables you to decide later where to print it. Use SQR Viewer or SQR Print to print an SPF file.

Using the DECLARE-PRINTER Command

The DECLARE-PRINTER command specifies printer-specific settings for the output file types that SQR supports: line printer, PostScript, HP LaserJet, and HTML. The DECLARE-PRINTER command itself does not cause the report to be produced for a specific printer. To specify a specific format, use one of these methods:

- The -PRINTER:*xx* command-line flag.

  For example -PRINTER:PS produces PostScript output. If the program creates multiple reports, such as the sample program ex18a.sqr, the -PRINTER:*xx* flag produces the same output format for all of the reports.

- The USE-PRINTER-TYPEcommand.

  You must use this command before you print because SQR cannot switch the printer type in the middle of a program. USE-PRINTER-TYPE PS, for example, produces PostScript output.

- The PRINTER-TYPE option of the DECLARE-REPORT command.

  You normally use the DECLARE-REPORT command when a program generates more than one report.

For example, the following code example produces PostScript output for the labels report:

```
declare-report labels
    layout=labels
    printer-type=ps
end-declare
```

The DECLARE-PRINTER command defines settings for line printers, PostScript, or HP LaserJet printers. Specify the type of printer by using the *type* option of the DECLARE-PRINTER command or one of the predefined printers: DEFAULT-LP, DEFAULT-PS, DEFAULT-HP, and DEFAULT-HT.

A program can have more than one DECLARE-PRINTER command if you define settings for each of the printer types. The settings for a particular printer take effect only when output is produced for that printer. When the program generates multiple reports, you can define settings for each printer for each report. To make a DECLARE-PRINTER command apply to a specific report, use the FOR-REPORTS option.

The output file normally has the same name as the program, but with a different file extension. The default file extension is .lis for PostScript (PS), HP LaserJet (HP), or Line Printer (LP). If you are generating an SPF, the default extension is .spf. If you want SQR to use another name for the output file (including a user-defined file extension), use the -F option on the command line. For example, to use chapter1.out as the output of the sample program ex1a.sqr, use this command to run SQR:

```
sqr ex1a username/password -fchapter1.out
```


When a program creates more than one report, you can name the output file by using multiple -F flags:

```
sqr ex20a username/password -flabel.lis -fletter.lis -flisting.lis
```

You cannot directly name .spf files. You can still use the -F command-line flag to name the file, but you cannot control the file name extension. For example:

```
sqr ex20a username/password -flabel.lis -fletter.lis -flisting.lis -nolis
```

The -NOLIS command-line flag causes SQR to produce .spf files instead of .lis files. The actual file names are label.spf, letter.s01, and listing.s02. The second .spf file is named .s01 and the third is named .s02. SQR supplies file extensions such as these when a program generates multiple reports.

Different operating systems require different techniques for printing output. On platforms other than Microsoft Windows, if output is in SPF format, you first use SQR Print to create the printer-specific file. For example, the following command invokes SQR Print to create a PostScript file named myreport.lis from the output file named myreport.spf:

```
sqrp myreport.spf -printer:ps
```

This conversion is one-way—an .spf file can be converted to an .lis file, but an .lis file cannot be converted to an .spf file.

The following table summarizes the commands and command-line options that you can use on different systems to send report output to a printer. Consult your operating system documentation for details.

Operating System	***Command***	***Command-Line Options***
UNIX	lp myreport.lis lp myreport.lis -d ...	Use -D for printer destination. You can use the UNIX *at* command to schedule the printing time.
Microsoft Windows	SQR prints directly. You can also use SQR Viewer.	Use the Print Setup dialog box in SQR Print or the SQR Viewer to select a printer destination. Use SQR Print to print multiple copies. You can also use the File Manager Copy command to copy the file to the printer destination (for example, lpt1).

Check with your systems administrator about other procedures or commands that are applicable to printing output files at your site.

Related Links

Understanding the Sample Program for Multiple Reports

Chapter 28

Using the SQR Command Line

Understanding the SQR Command Line

You can use the SQR command line to specify flags and to pass arguments to modify your program at runtime.

You can enter command-line flags such as -Bnn, -KEEP, or -S in the command line to modify some aspect of program processing or output. Command-line arguments are typically answers to requests (done in the SQR program by ASK or INPUT commands) for user input.

The following code example and table describe the syntax of the SQR command line:

```
SQR [program] [connectivity] [flags ...] [args ...] [@file ...]
```

Argument	***Description***
program	The name of the program. The default file type or extension is .sqr. If the parameter is entered as a question mark (?) or omitted, SQR prompts you for the program name. On UNIX/Linux-based systems, if your shell uses the question mark as a wildcard character, you must precede it with a backslash (\).
connectivity	Oracle: Use [Username]/[Password[@Database]] as your username and password for the database. You can also specify the connection string for the database (for example, @B:ORASERVER). The information that SQR needs to connect to the database. If the parameter is entered as a question mark or omitted, SQR prompts you for it. The information you enter depends on the database you are using: DB2: Use Ssname and SQLid for the subsystem name and SQL authorization ID. Informix: Use Database as the name of the database. ODBC: Use Data_Source_Name/[Username]/[Password] as the name of the ODBC driver when you set up the driver and your username and password for the database.
flags	Any of the flags that are listed in the SQR Language Reference. Begin command-line flags with a hyphen. When a flag has an argument, enter the argument directly following the flag with no intervening space. See "SQR Command-Line Flags" (PeopleTools 8.58: SQR Language Reference for PeopleSoft).

Argument	*Description*
args...	Arguments that are used by SQR while the program is running. Arguments that are listed here are used by the ASK and INPUT commands rather than prompting the user. Arguments must be entered in the command line in the same sequence that they are expected by the program: first all ASK arguments in order and then INPUT arguments in order.
@file...	File containing program arguments, one argument per line. Arguments listed in the file are processed one at a time. You can specify the command-line arguments program, connectivity, and args in this file.

Specifying Command-Line Arguments

This section provides an overview of command-line arguments and discusses how to:

- Retrieve arguments.
- Specify arguments and argument files.
- Use an argument file.
- Use other approaches to pass command-line arguments.
- Use reserved characters.
- Create an argument file from a report.

Understanding Command-Line Arguments

You can pass an almost unlimited number of command-line arguments to SQR at runtime. On some platforms, the operating system imposes a limit on the number of arguments or the total size of the command line. Passing arguments is especially useful in automated reports, such as those that are invoked by scripts or menu-driven applications.

You can pass arguments to SQR on the command line, in files, or with the SQRFLAGS environment variable. When you pass arguments in a file, reference the file name on the command line and put one argument on each line of the file. This avoids any limits that are imposed by the operating system.

To reference a file on the command line, precede its name with the @ sign as shown in the following code example:

```
sqr myreport sammy/baker arg1 arg2 @file.dat
```

In this example, *arg1* and *arg2* are passed to SQR, followed by the file.dat file. Each line in file.dat has an additional argument.

Retrieving Arguments

When the ASK and INPUT commands run, SQR determines whether you entered any arguments in the command line or whether an argument file was opened. If either has happened, SQR uses this input

instead of prompting the user. After the available arguments are used, subsequent ASK or INPUT commands prompt the user for input. If you use the INPUT command with the BATCH-MODE argument, SQR does not prompt the user but instead returns a status meaning *No more arguments.*

SQR processes all ASK commands before INPUT commands.

Note: If you compiled the SQR program into an .SQT file, ASK commands will already have been processed. Use INPUT instead.

Specifying Arguments and Argument Files

You can mix argument files with simple arguments, as shown in the following code example:

```
sqr rep2 sammy/baker 18 @argfile1.dat "OH" @argfile2.dat "New York"
```

This command line passes SQR the number 18, the contents of argfile1.dat, the value OH, the contents of argfile2.dat, and the value New York, in that order.

The OH argument is in quotes to ensure that SQR uses uppercase OH. When a command-line argument is case-sensitive or contains spaces, you must enclose it within quotes. Arguments that are stored in files do not require quotes and cannot contain them; the actual strings with uppercase characters and any spaces are passed to SQR.

Using an Argument File

To print the same report on different printers with different characteristics, you can save values for the different page sizes, printer initializations, and fonts in separate files and use a command-line argument to specify which file to use. For example, the following command line code example passes the value 18 to SQR:

```
sqr myreport sammy/baker 18
```

An #INCLUDE command in the report file selects the printer18.dat file based on the command-line argument:

```
begin-setup
  ask num      ! Printer number.
  #include 'printer{num}.dat'  ! Contains #DEFINE commands for
               ! printer and paper width and length
  declare-layout report
        paper-size =({paper_width} {paper_length})
  end-declare
end-setup
```

In this example, the ASK command assigns the value 18 to the *num* variable; 18 is a compile-time argument. The #INCLUDE command then uses the value of *num* to include the printer18.dat file, which could include commands like this:

```
! Printer18.dat-definitions for printer in Bldg 4.
#define paper_length 11
#define paper_width 8.5
#define bold_font LS12755
#define light_font LS13377
#define init HM^J73011
```


Using Other Approaches to Pass Command-Line Arguments

SQR examines an argument file for a program name, username, or password if none is provided in the command line. The following command line omits the program name, username, and password:

```
sqr @argfile.dat
```

The first two lines of the argument file for this code example contain the program name and the username and password:

```
myreport
sammy/baker
18
OH
...
```

If you do not want to specify the report name, username, or password in the command line or in an argument file, use the question mark (?). SQR prompts the user to supply these. For example:

```
sqr myreport ? @argfile.dat
```

In this example, the program prompts the user for the username and password instead of taking them from the first line in the argument file.

You can use more than one question mark on the command line, as shown in the following code example:

```
sqr ? ? @argfile.dat
```

In this example, the program prompts the user for the program name and the username and password.

Note: SQR for Microsoft Windows does not accept the SQR program name and database connectivity to be part of the argument file.

Using Reserved Characters

The hyphen (-) and @ sign characters have special meanings in a command line. The hyphen precedes an SQR flag, and the @ sign precedes an argument file name. To use either of these characters as the first character of a command-line argument, enter the character twice to indicate that it is a literal hyphen or @ sign, as shown in the following code example:

```
sqr myreport ? --17 @argfile.dat @@X2H44
```

In this example, the double hyphen and double @ sign are interpreted as single literal characters.

Creating an Argument File from a Report

You can create an argument file for one program from the output of another program. For example, you can print a list of account numbers to the acctlist.dat file, and then run a second report with the following command:

```
sqr myreport sammy/baker @acctlist.dat
```

End acctlist.dat with a flag such as END, as shown in the following code example:

```
123344
134455
156664
 ...
```



```
END
```

An SQR program can use the numbers in acctlist.dat with an INPUT command, as shown in the following code example:

```
begin-procedure get_company
next:
input $account            batch-mode status = #status
  if #status = 3
      goto end_proc
  end-if
begin-select
cust_num, co_name, contact, addr, city, state, zip
  do print-page               ! Print page with
               ! complete company data
from customers
where cust_num = $account
end-select
goto next               ! Get next account number
end_proc:
end-procedure !get_company
```

Using Batch Mode

SQR enables you to run reports in batch mode in:

- UNIX/Linux.
- Microsoft Windows.

You can create UNIX/Linux shell scripts or MS-DOS batch (.bat) files to run SQR. Include the SQR command line in the file as you enter it.

Chapter 29

Generating and Publishing HTML from an SQR Program

Understanding SQR Capabilities That Are Available with HTML

The SQR language has a rich set of features, but some of these features are not available for HTML output because of the limitations of that format.

The SQR features that are supported for HTML include:

- Images.
- Font sizing.

  The SQR language specifies font sizes in points. HTML specifies font sizes in a value from 1 to 6. A point size that is specified in an SQR program is mapped to an appropriate HTML font size.
- Font styles.

  The bold and underline font styles are supported.
- Centering.

The SQR features that are not currently supported for HTML output include:

- Font selection.
- Bar codes.
- Lines and boxes (using -PRINTER:HT).

Note: You can generate professional quality HTML report files with SQR without being an HTML expert. However, if you want to adapt HTML output by using the SQR HTML procedures, you may want to learn more about HTML.

Generating HTML Output

This section provides an overview of HTML output and discusses how to:

- Produce HTML output.
- Use -PRINTER:EH.
- Set HTML attributes under -PRINTER:EH.

- Use -PRINTER:HT.
- Burst reports.
- Set attributes with HTML procedures.
- Use additional HTML procedures.
- Set output file types.
- Test HTML output.

Understanding HTML Output

When an SQR program generates HTML output, that output contains HTML tags. An HTML tag is a character sequence that defines how information appears in a web browser.

Typically, HTML output looks like this:

```
<HTML><HEAD><TITLE>myreport.lis</TITLE></HEAD><BODY>
```

This code is only a portion of the HTML output that SQR generates. The tags that it contains indicate the start and end points of HTML formatting.

For example, in the code example, the <HTML> tag identifies the output that follows as HTML output. The <TITLE> and </TITLE> tags enclose the report title, in this case, myreport.lis. The <BODY> tag indicates that the information following it makes up the body of the report.

Producing HTML Output

You can produce HTML output from an SQR program by using one of four methods, each of which provides a different level of HTML features:

- Running an unmodified SQR program with the -PRINTER:EH command-line flag makes the HTML 3.0 or 3.2 output viewable in a web browser.
- Running an unmodified SQR program with the -PRINTER:HT command-line flag makes the HTML 2.0 output viewable in a web browser.
- Using two HTML procedures, html_set_head_tags and html_set_body_attributes, enables you to define a title and background image for HTML output.

  With this method, you must still use the -PRINTER:HT command-line flag.
- Using additional HTML procedures produces output with a full set of HTML features, including lists, tables, and links.

  With this method, you must still use the -PRINTER:HT command-line flag.

The procedures that are used in the last two options are contained in a file called html.inc. To use HTML procedures, the SQR program must include this command:

```
#include 'html.inc'
```

The HTML.INC file is located in the SAMPLE (or SAMPLEW) directory. Use the -I command-line flag to specify its path.

Using -PRINTER:EH

You can generate enhanced HTML output from an SQR program by using the -PRINTER:EH command-line flag. Output that contains HTML formatting tags is produced. All output is displayed as fully formatted HTML 3.0 or 3.2 text. You can generate high-quality HTML from SQR programs by using -PRINTER:EH to issue a command like this:

```
sqrw myreport.sqr sammy/baker@rome -PRINTER:EH
```

You can control the version of HTML that is used by editing the FullHTML enhanced HTML parameter in the PSSQR.INI file. Set FullHTML to be equal to TRUE for HTML 3.2 or FALSE for HTML 3.0. Adjust this setting based on the level of HTML that your web browser supports. The -PRINTER:EH default output is HTML 3.0.

If you have existing .spf files for which you want to generate enhanced HTML output, you do not need to rerun your SQR program. You can invoke SQR Print (with SQRP or SQRWP, depending on your platform) to generate enhanced HTML from .spf files by using a command like this:

```
sqrwp myreport.spf -PRINTER:EH
```

From within the SQR Viewer, you can also generate the same high-quality HTML by selecting File, Save as HTML. The HTML level output from the SQR Viewer is also determined by the PSSQR.INI file settings and has the same default value.

You can also generate enhanced HTML files with precompiled SQR program files (.sqt files). Run the .sqt file against SQR Execute with a command like this:

```
sqrwt myreport.sqt sammy/baker@rome -PRINTER:EH
```

As is true when running any .sqt file, you can run it against SQR (or sqrw on Microsoft Windows platforms) by including the -RT flag. To generate enhanced HTML, use the -PRINTER:EH flag in the command:

```
sqrw myreport.sqr sammy/baker@rome -RT -PRINTER:EH
```

The sample program ex7a.sqr produces a simple master and detail report. By running it with -PRINTER:EH, you can produce HTML output. A left frame is produced with links to each page of the report. The right frame features a navigation bar that appears at the top of every page in the report. The navigation bar enables you to move to the first or last page or to move one page forward or backward from your relative page viewing position.

With -PRINTER:EH, you can also use additional flags to modify the output, such as:

- -EH_CSV

  This flag creates an additional output file in comma separated values (CSV) format.

- -EH_CSV:*file*

  This flag associates the CSV icon with the specified file.

- -EH_Icons:*dir*

  This flag specifies the directory in which the HTML should find the referenced icons.

- -EH_Scale:{*nn*}

This flag sets the scaling factor from 50 to 200.

These flags work only with -PRINTER:EH.

Setting HTML Attributes Under -PRINTER:EH

In certain cases, you may want additional control over the enhanced HTML code that is generated with -PRINTER:EH. SQR supports extensions that enable you to control the generated HTML by specifying titles, background colors and images, links, text colors, and more.

Specifying HTML Titles

The HTML page title normally appears on the caption bar of the browser window and is also used when you are creating a bookmark for the page. It is placed between the <TITLE> and </TITLE> HTML tags. Specify the title of the HTML page by using the %%Title extension at the beginning of the SQR program by entering:

```
Print-Direct Printer=html '%%Title Monthly Sales'
```

Specifying Background Colors

Specify a background color for the pages that are generated with -PRINTER:EH by using the %%Body-BgColor extension. Enter code like this at the beginning of the program:

```
Print-Direct Printer=html '%%Body-BgColor #0000FF'
```

To set the background color for the navigation bar, enter code like this:

```
Print-Direct Printer=html '%%Nav-Body-BgColor #0000FF'
```

See “Specifying HTML Colors.”

Specifying Background Images

To use a background image for the report pages that the enhanced HTML generates, insert the % %Background extension at the beginning of the program:

```
Print-Direct Printer=html '%%Background tile.gif'
```

To set the background image for the navigation bar, enter code like this:

```
Print-Direct Printer=html '%%Nav-Background D:\jpegdir\house.jpg'
```

The background attribute can be any valid URL. If you do not specify the %%Nav-Background extension while specifying the body background, the background image that you specify for the body is used both in the body and in the navigation bar. If you do not want an image to appear in the navigation bar, use code like this:

```
Print-Direct printer=html '%%Nav-Background EMPTY'
```

Specifying Links

The %%Href extension specifies a link in the report. This extension enables you to make a text, number, image, or chart object into a link. The object can be the item that you click to activate the link or it can be

the location on the page where the link takes you. Specify the latter by using the %%Anchor extension. For example:

```
Print-Direct Printer=html '%%Href #section2'
Print 'ABC' ()
...
Print-Direct Printer=html '%%Anchor section2'
Print 'XYZ' ()
```

In this example, clicking the ABC text on the page jumps to the XYZ text. When using frames or multiple browser windows, you can control which frame displays the target of the link by using the target option of the %%Href extension. For example, specify on one line:

```
Print-Direct Printer=html '%%Href target="_top" http://www.example.com'
```

Specifying Text Colors

Use the %%Color and %%ResetColor extensions to change the color of text. The following code example demonstrates this capability:

```
If &Salary > 100000
Print-Direct Printer=html '%%Color #FF0000'
End-If
Print &Salary ()
If &Salary > 100000
Print-Direct Printer=html '%%ResetColor'
End-If
```

In this example, when the value of the column is more than 100,000, it prints in red. The %%Color extension affects all text (and number) printing from this point on. This behavior is similar to that of the ALTER-PRINTER command. A subsequent invocation of %%Color with a different color value sets the current color to a new color. To restore the color to the default (normally, black) use the %%ResetColor extension.

Specifying HTML Colors

Specifying color as a red-green-blue (RGB) hexadecimal value is the only way to designate color in SQR. Your browser documentation should contain a listing of supported colors and their hexadecimal values. To specify color as an RGB hexadecimal value, enter a # character followed by six hexadecimal digits. The first two digits specify the intensity of the red, the next two specify the green, and the last two specify the blue. For example, green is #00FF00.

Including Your Own HTML Tags

Enhanced HTML extensions enable you to include your own HTML tags in the output. These tags are passed through to the output without change. Use this feature to include advanced HTML capabilities such as JavaScript and <APPLET> tags.

SQR PRINT with CODE-PRINTER=HT enables you to inject any text into the HTML output. SQR does not check the text that you are printing. This text can contain anything that your browser understands. Do not use this method for formatting, because your formatting may conflict with -PRINTER:EH-enhanced HTML formatting. -PRINTER:EH-enhanced HTML uses HTML tables extensively. To fully control the formatting, use the HTML procedures that are defined in html.inc and that are documented in this section. By invoking the html_on procedure, you instruct the enhanced HTML to perform no formatting. Specify all formatting by using the HTML procedures in html.inc or by using SQR PRINT with CODE-PRINTER=HT to insert HTML code. When you use SQR PRINT with CODE-PRINTER=HT, the enhanced HTML does not translate special symbols that are used in HTML tags, such as <, >, and &.

Related Links

SQR Language Reference for PeopleSoft

Using -PRINTER:HT

Another method for generating HTML output from an SQR program is running a program with the command-line flag -PRINTER:HT. Alternatively, you can make some simple modifications to the program. Add either DECLARE-PRINTER with the TYPE=HT argument or USE-PRINTER-TYPE HT.

With these methods, HTML output is generated in the following way:

- All output appears as preformatted text by using the <PRE> and </PRE> HTML tags.
- Text appears on the page at the position coordinates that are specified in the SQR program.
- Text appears in a fixed-width font, such as Courier.
- Font sizes map to HTML font sizes.
- HTML reserved characters map to the corresponding HTML sequence.

  The <, >, &, and " characters map to the <, >, &, and ", character sequences, respectively, thus preventing the web browser from mistaking such output as an HTML sequence.

The sample program ex7a.sqr produces a simple master and detail report. By running it with -PRINTER:HT, you can produce HTML output. A left frame is produced with links to each page of the report. The right frame features a navigation bar that appears at the top of every page in the report. The navigation bar enables you to move to the first or last page or to move one page forward or backward from your relative page viewing position.

See Using the DECLARE-PRINTER Command and Understanding Printer-Independent Reports.

Bursting Reports

With SQR, you can generate HTML format reports by using -PRINTER:EH or -PRINTER:HT command-line flags. If you want HTML files to be smaller in size for faster load times or to be divided on the basis of report page ranges, or if you want to preview the table of contents for a report in your web browser without generating an entire report, use -BURST:{*xx*} with -PRINTER:EH or -PRINTER:HT.

By using -BURST:P (or BURST:P1) with -PRINTER:EH or by using -BURST:P1 with -PRINTER:HT, you can generate HTML output files that are *burst* by report page numbers, one report page per .htm file. (This practice is frequently referred to as demand paging.) As a result, a 25-page report would be divided into 25 separate .htm output files. By using -PRINTER:HT, you can also specify the report page ranges that you want within an HTML file. For example, -BURST:P0,1,3-5 generates an HTML file containing only report page numbers 1, 3, 4, and 5. You can then focus on information that is truly of interest.

Similarly, if you specify -PRINTER:HT with -BURST:T, only the table of contents file is generated. And if you specify -PRINTER:HT with -BURST:S, report output is generated according to symbolic table of contents entries. By using -BURST:S, you can specify the numeric level to burst on (for example, -BURST:S2 bursts on level 2). If you have used DECLARE-TOC and TOC-ENTRY commands in the SQR program, the table of contents provides more detailed information than just page number links, as illustrated by the following code example.

To use DECLARE-TOC and TOC-ENTRY to improve the information that is available in generated HTML output, this example adds the following code example to the beginning of the sample program ex7a.sqr:

```
begin-setup
declare-toc common
       for-reports=(all)
       dot-leader=yes
       indentation=2
end-declare
end-setup
```

The code example also adds this code to the body of the program, in the main procedure immediately following the begin-select and Print ‘Customer Information’ (,1):

```
toc-entry text = &name
```

Setting Attributes with HTML Procedures

Use the SQR HTML procedures html_set_head_tags and html_set_body_attributes to define a title and background image for a report. To use these procedures, the SQR program must include the html.inc file. You must also run the program by using the -PRINTER:HT command-line flag.

These procedures must be called at the start of the program. For example:

```
do html_set_head_tags('<TITLE>Monthly Report</TITLE>')
do html_set_body_attributes('BACKGROUND="/images/mylogo.gif"')
```

The first line of this code example displays the *Monthly Report* title. Specifically, the entire '<TITLE>Monthly Report</TITLE>' sequence is passed as an argument to the html_set_head_tags procedure. The argument is enclosed in single quotes.

The second line displays the mylogo.gif background image for the web page. Again, an argument is passed to the procedure. The entire argument is enclosed in single quotes, and the file name and path are enclosed in double quotes.

Together, these two lines of code generate the following HTML output:

```
<HTML><HEAD><TITLE>Monthly Report</TITLE></HEAD>
<BODY BACKGROUND="/images/mylogo.gif">
```

Using Additional HTML Procedures

Using additional HTML procedures in the SQR program provides enhanced capabilities, including:

- Highlighting , including HTML physical tags and logical markup tags.

  HTML physical tags include subscript, superscript, and strikethrough. HTML logical markup tags include citation, code, keyboard, and sample.

- Headings.
- Links.
- Lists , including ordered lists, unordered lists, definition lists, directory lists, and menus.
- Paragraph formatting , including paragraph breaks, line breaks, and horizontal dividers.

- Tables , including captions, rows, columns, and column headings.

Setting Output File Types

An SQR report named myreport.sqr creates a FRAME file (myreport.htm) and report output files. The OUTPUT-FILE-MODE entry in the Default-Setting section of the PSSQR.INI file controls the report output file extensions. When this entry is set to SHORT, the report output files use the form myreport.hzz, and when set to LONG, the files use the form myreport_zz.htm. The value of *zz* ranges from 00 to 99 and reflects the report number.

The FRAME file displays a list (links) of report pages in one frame and the report text in another frame. Each report output file contains a list of pages (links) at the end of the file. If myreport.sqr created multiple reports, then the FRAME file contains a link to each report output file. In addition, each report output file contains links to the other report output files that were created during the program run.

Testing HTML Output

When an SQR program produces HTML output, you can preview it on a local system. This is a good way to test the output before you publish it on a website.

To test the output of the program, open the file in the web browser. If your web browser supports the HTML FRAME construct, open the FRAME file (myreport_frm.htm); otherwise, open the report output file (myreport.h00, myreport_00.htm).

Using HTML Procedures in an SQR Program

This section provides an overview of HTML procedures and discusses how to:

- Use HTML procedures.
- Position objects.
- Display records in tables.
- Create headings.
- Highlight text.
- Create links.
- Include images.
- Display text in lists.
- Format paragraphs.
- Incorporate your own HTML tags.

Related Links

SQR Language Reference for PeopleSoft

Understanding HTML Procedures

To enhance the appearance of HTML output, use HTML procedures in an SQR program.

An SQR program with these procedures generates output as described previously in "Using PRINTER:HT," with these exceptions:

- The <PRE> and </PRE> HTML tags are not used.
- Text appears in a proportional font, such as Arial.
- Positioning values that are specified in the SQR program are ignored.

  Text, HTML tags, and other information are placed in the HTML output in the order in which they are generated by the SQR program.
- White space, such as spaces between PRINT commands, is removed.

Using HTML Procedures

When using the HTML procedures, include the html.inc file. As before, you must run the SQR program with the -PRINTER:HT command-line flag.

The SQR program must also call the html_on procedure at the start of the program. The command that calls this procedure is:

```
do html_on
```

Additionally, the program must specify a large page length to prevent page breaks. SQR automatically inserts the page navigation links and an <HR> HTML tag at a page break. If a page break occurs in the middle of an HTML construct, such as a table, the output can appear incorrectly. Use the DECLARE-LAYOUT command with a large MAX-LINES setting to prevent page breaks from occurring.

Positioning Objects

When HTML procedures are activated:

- HTML output is generated without the <PRE> and </PRE> tags.
- All position qualifiers in the SQR program are ignored, and program output and HTML tags are placed in the output file in the order in which they are generated, regardless of their position qualifiers.
- The text that is printed in a BEGIN-HEADING section does not appear at the top of a page.

  Because no positioning is done, text in the heading appears at the bottom.
- White space, such as spaces between PRINT commands, is removed.

Thus, you must use the HTML procedures to format the report.

The following code example does not use the HTML procedures to format the output:

```
print 'Report summary:' (1,1)
print 'Amount billed:' (3,1)
print #amount_amount (3,20)
```



```
print 'Total billed:' (4,1)
print #total_amount (4,20)
```

In this case, all of the text appears on the same line and with no spaces between the data.

With the HTML procedures for line breaks and a table, you can format the output properly.

The following code example uses the html_br procedure to separate the first two lines of text. The html_table, html_tr, html_td, and html_table_end procedures display the totals in a tabular format. An empty string is passed to each procedure as it is called. This empty string is required if no other argument is passed.

```
print 'Report summary:' (1,1)
do html_br(2,'')
do html_table('')
do html_tr('')
do html_td('WIDTH=300')
print 'Amount billed:' (3,1)
do html_td('')
print #amount_amount (3,20)
do html_tr('')
do html_td('WIDTH=300')
print 'Total billed:' (4,1)
do html_td('')
print #total_amount (4,20)
do html_table_end
```

Displaying Records in Tables

When HTML procedures are activated, all positioning values in the SQR program are ignored. Thus, the position values cannot be used to display records in a tabular format. To display records in a tabular format, use the following procedures:

Description	***Beginning Procedure***	***End Procedure***
Create a table.	html_table	html_table_end
Create a caption. The end is typically implied and html_caption_end is not required, but you can use it for completeness.	html_caption	html_caption_end
Create rows. The end is typically implied and html_tr_end is not required, but you can use it for completeness.	html_tr	html_tr_end
Create column headings. The end is typically implied and html_th_end is not required, but you can use it for completeness.	html_th	html_th_end
Create columns. The end is typically implied and html_td_end is not required, but you can use it for completeness.	html_td	html_td_end

The following sample program uses these table procedures to display information in a tabular format:

```
Program ex28a.sqr
#include 'html.inc'
begin-program
```



```
   do main
end-program
! set a large page length to prevent page breaks
begin-setup
  declare-layout default
    max-lines=750
  end-declare
end-setup
begin-procedure main
! turn on HTML procedures
 do html_on
! start the table and display the column headings
do html_table('border')
do html_caption('')
print 'Customer Records' (1,1)
do html_tr('')
do html_th('')
print 'Cust No' (+1,1)
do html_th('')
print 'Name'    (,10)
! display each record
begin-select
   do html_tr('')
   do html_td('')
cust_num         (1,1,6) edit 099999
   do html_td('')
name          (1,10,25)
   next-listing skiplines=1 need=1
from customers
end-select
! end the table
do html_table_end
end-procedure
```

Creating Headings

The heading procedures display text by using heading levels like those in a book. The available heading levels range from 1 to 6; a first-level heading is the highest. To use the heading procedures, call the appropriate heading procedure before the text is generated. After the text is generated, call the corresponding end procedure.

The following code example displays text as a second-level heading:

```
do html_h2('')
print 'A Level 2 Heading' (1,1)
do html_h2_end
```

Highlighting Text

The highlighting procedures enable you to display text in the various HTML highlighting styles. Highlighting is also called logical markup.

To use the highlighting procedures, call the appropriate highlighting procedure before the text is generated. After the text is generated, call the corresponding end procedure.

The following highlighting procedures are available:

Type of Highlighting	***Beginning Procedure***	***End Procedure***
Blink	html_blink	html_blink_end
Citation	html_cite	html_cite_end

Type of Highlighting	*Beginning Procedure*	*End Procedure*
Code	html_code	html_code_end
Keyboard	html_kbd	html_kbd_end
Sample	html_sample	html_sample_end
Strike	html_strike	html_strike_end
Subscript	html_sub	html_sub_end
Superscript	html_sup	html_sup_end

The following code example displays text in the subscript style:

```
print 'Here is ' (1,1)
do html_sub('')
print 'subscript' ()
do html_sub_end
print ' text' ()
```

Creating Links

The link procedures enable you to create links and link anchors. When a user clicks a link, the web browser switches to the top of the specified HTML document, to a point within the specified document, or to a link anchor within the same document. A link can point to the home page of a website, for example.

To insert a link, use the html_a procedure to format the information that is to become the link, and use the html_a_end procedure to mark the end of the link. Two useful attributes for the html_a procedure are the HREF and NAME attributes:

- Use the HREF attribute to specify the location to which the link points.
- Use the NAME attribute to specify an anchor to which a link can point.

These attributes are passed as arguments to the html_a procedure.

The following code example creates an anchor and two links. The anchor is positioned at the top of the document. The first link points to the HTML home.html document. The second link points to the anchor named TOP in the current document. Note the # sign in the argument, which indicates that the named anchor is a point within a document. The third link points to an anchor named POINT1 in the mydoc.html document.

```
do html_a('HREF=home.html')
print 'Goto home page' ()
do html_a_end

do html_a('NAME=TOP')
do html_a_end

print 'At the top of document' ()
do html_br(40, '')
print 'At the bottom of document' ()
do html_p('')

do html_a('HREF=#TOP')
print 'Goto top of document' ()
do html_a_end
```



```
do html_a ('HREF=mydoc.html#POINT1')
print 'Goto point1 in mydoc.html' ()
do html_a_end
```

Including Images

You can include an image in an HTML output with the PRINT-IMAGE command or the html_img procedure. Both of these produce the <IMG> HTML tag.

The PRINT-IMAGE command displays images for all printer types but enables you to specify only the image type and source. The html_img procedure displays images only for the HTML printer type, but it enables you to specify any of the attributes that are available for an <IMG> HTML tag.

For HTML output, you can use only Graphics Interchange Format (GIF) or JPEG files. With PRINT-IMAGE, use the TYPE=GIF-FILE or TYPE=JPEG-FILE argument, respectively.

Displaying Text in Lists

The list procedures display lists. To use these procedures, call the appropriate procedure before the list is generated. After the list is generated, call the corresponding end procedure.

The following list procedures are available:

List Type	***Beginning Procedure***	***End Procedure***
Definition (terms and their definitions)	html_dl	html_dl_end
Directory	html_dir	html_dir_end
Menus	html_menu	html_menu_end
Ordered (numbered or lettered)	html_ol	html_ol_end
Unordered (bulleted)	html_ul	html_ul_end

To display a list, except for the definition list, call the appropriate list procedure before starting the output. Call html_li to identify each item in the list; you can also call html_li_end for completeness. After specifying the output, call the corresponding end procedure.

The following code example displays an ordered list:

```
do html_ol('')
do html_li('')
print 'First item in list' (1,1)
do html_li_end
do html_li('')
print 'Second item in list' (+1,1)
do html_li_end
do html_li('')
print 'Last item in list' (+1,1)
do html_li_end
do html_ol_end
```

To display a definition list, call html_dl before starting the output. Call html_dt to identify a term and html_dd to identify a definition. After specifying the output, call html_dl_end. You can also call html_dd_end and html_dt_end for completeness.

The following code example displays a definition list:

```
do html_dl('')
do html_dt('')
print 'A daisy' (1,1)
do html_dt_end
do html_dd('')
print 'A sweet and innocent flower' (+1,1)
do html_dd_end
do html_dt('')
print 'A rose' (+1,1)
do html_dt_end
do html_dd('')
print 'A very passionate flower' (+1,1)
do html_dd_end
do html_ol_end
```

Formatting Paragraphs

The HTML procedures provide various paragraph formatting capabilities. To use these procedures, call the appropriate paragraph procedure before the list is created.

The following procedures are available:

Formatting Type	*Beginning Procedure*	*End Procedure*
Paragraph break	html_p	html_p_end Many HTML constructs imply an end of paragraph; thus, the html_th_end procedure is not needed, but you can use it for completeness.
Line break	html_br	NA
Horizontal divider (usually a sculpted line)	html_hr	NA
Prevent text wrapping	html_nobr	html_nobr_end

The following code example uses the paragraph formatting procedures to format text into paragraphs:

```
print 'Here is some normal text' (1,1)
do html_p('ALIGN=RIGHT')
print 'Here is right aligned text' (+1,1)
do html_br(1,'')
print 'and a line break' (+1,1)
do html_p_end
do html_hr('')
do html_nobr('')
print 'A very long line of text that cannot be wrapped' (+1,1)
do html_nobr_end
```

Incorporating Your Own HTML Tags

You can incorporate your own HTML tags into the HTML output. To do so, use the PRINT command with the CODE-PRINTER=HT argument.

Text that is printed with this argument is placed only in the HTML output that is generated when the HTML printer type is specified. With all other printer types, the text is not placed in the output. In

addition, the specified text is placed directly in the HTML output without any modifications, such as the mapping of reserved characters.

The following code example uses the <B> HTML tag to print bold text:

```
print '<B>' () code-printer=ht
print 'Bold text' ()
print '</B>' () code-printer=ht
```

Modifying an Existing SQR Program for HTML

In this section, an existing sample program, ex12a.sqr, was modified to use HTML procedures. The modified program is named program ex28b.sqr. First, examine the output from ex12a.sqr when this program is run without modifications by using the -PRINTER:HT command-line flag. Three HTML files are generated: ex12a.htm, ex12a_frm.htm, and ex12a_toc.htm.

```
Program ex28b.sqr
#include 'html.inc'
begin-setup
  declare-layout default
    max-lines=10000
  end-declare
end-setup
begin-program
 do main
end-program
begin-procedure main
do html_on
print $current-date (1,1) edit 'DD-MON-YYYY'
do html_p('')
do html_table('BORDER')
do html_tr('')
do html_th('WIDTH=250')
print 'Name'   (3,1)
do html_th('WIDTH=120')
print 'City'   (,32)
do html_th('WIDTH=60')
print 'State'  (,49)
do html_th('WIDTH=90')
print 'Total'  (,61)
begin-select
  do html_tr('')
  do html_td('')
name  (,1,30)
  do html_td('')
city  (,+1,16)
  do html_td('')
state (,+1,5)
  do html_td('ALIGN=RIGHT')
tot   (,+1,11) edit 99999999.99
  next-listing no-advance need=1
  let #grand_total = #grand_total + &tot
from customers
end-select
  do html_tr('')
  do html_tr('')
  do html_td('COLSPAN=3 ALIGN=RIGHT')
print 'Grand Total' (+1,40)
  do html_td('ALIGN=RIGHT')
print #grand_total (,55,11) edit 99999999.99
do html_table_end
end-procedure ! main
```


In this code example, a DECLARE-LAYOUT command with a large page length setting that is specified in the MAX-LINES argument is issued to prevent page breaks.

The html_on procedure activates the HTML procedures.

The html_table, html_tr, html_td, and html_th procedures position the information in a tabular format. Note the arguments that are passed to the HTML procedures:

- BORDER produces the sculpted border.
- WIDTH defines the width of the columns.
- ALIGN right-aligns the text in the Total column.
- COLSPAN causes the Grand Total label to be spanned beneath three columns of data.

Instead of using a HEADING section, use the html_tr and html_th procedures to display column headings.

See Displaying Records in Tables.

Publishing a Report

This section discusses how to:

- Publish reports.
- Support older browsers.
- View published reports.
- Publish by using an automated process.
- Publish by using a Common Gateway Interface (CGI) script.

Publishing Reports

You can publish an SQR report on a website, and then anyone with a web browser can view the report over the internet or an intranet by specifying its URL.

To publish a report:

1. Run the SQR program.
2. Determine where the report output will be stored on the web server.

   The directory must be one that is referenced by a URL on the server. See your webmaster for more details about creating a URL.
3. Copy the generated HTML output files to the selected directory on the web server.

   If the output is generated on a client workstation, use a utility such as FTP to transfer the HTML output files to the web server.

Note: If you select the zip file option, a zip file is created for the generated HTML output in addition to the files being placed in the file system.

4. Create links on a home page or other website that point to the report files so that users browsing the network can navigate to the report and view it.

Supporting Older Browsers

To support older web browsers that do not support the HTML FRAME construct, create two separate links: one pointing to the FRAME file (.htm) and labeled to indicate the frame version, and another pointing to the report output file and labeled to indicate the nonframe version. If the report was created with HTML procedures, however, it should contain only a single page. In that case, a listing of report pages that are contained in the FRAME file is not needed. Only the report output file is required for publication on a website.

Viewing Published Reports

Use a web browser to view reports that are published on a website. To do this, specify a URL in your web browser, for example: http://www.myserver.com/myreport.htm.

Publishing by Using an Automated Process

The webmaster can create a program that automates the publishing process. The program should run the SQR program and copy the output to the appropriate location. You can start the program by using a scheduling utility to automatically run the program and publish it on the website at specified times.

The sample Bourne shell program:

- Sets the necessary environment variables.
- Runs the /usr2/reports/myreport.sqr program and generates the /usr2/reports/myreport.htm and /usr2/reports/myreport.h00 output files.
- Specifies /dev/null as the source of standard input to prevent the program from stopping if it requires input.
- Redirects the standard output to /usr2/reports/myreport.out to capture any status messages.

  You can view the output file at a later time to diagnose any problems.
- Copies the generated report files to the /usr2/web/docs directory to publish it on the web server.

  (Use the directory name that is appropriate for your server.)

Here is the code example:

```
#! /bin/sh
# set the appropriate environment values
ORACLE_SID=oracle7; export ORACLE_SID
ORACLE_HOME=/usr2/oracle7; export ORACLE_HOME
SQRDIR=/usr2/sqr/bin; export SQRDIR
# invoke the SQR program
sqr /usr2/reports/myreport.sqr orauser/orapasswd \
    -PRINTER:ht -I$SQRDIR \
```



```
    > /usr2/reports/myreport.out 2>&1 < /dev/null
# copy over the output
cp /usr2/reports/myreport.htm /usr2/web/docs
cp /usr2/reports/myreport.h00 /usr2/web/docs
```

Note: You must adjust the environment variables and the file names to fit your particular environment. See the documentation of your scheduling software for more details.

Publishing by Using a CGI Script

If you use the CGI script method, any user with a web browser can run an SQR and view the output. You can enable the user to run an SQR by providing a form to fill out.

When a user runs an SQR report through a website:

1. The user navigates to a form.
2. The user enters information on the form and clicks a button to invoke the CGI script.
3. The CGI script runs the SQR program.
4. The CGI script copies the report output file to the standard output.
5. The user views the report.

This process requires:

- The form
- The CGI script
- The SQR program

Creating the Form

Create an HTML form to enable the user to enter some values and start the request.

The following HTML code example defines a form with three radio buttons and a submit button. The radio buttons enable the user to specify the sorting criteria. The Submit button invokes the CGI script.

Here is the HTML code:

```
<HTML>
<TITLE>View Customer Information</TITLE>
<FORM METHOD=POST ACTION="/cgi-bin/myreport.sh">
<B>Select the Field to Sort By</B><P><DIR>
<INPUT TYPE="radio" NAME="rb1" VALUE="cust_num" CHECKED> Number<BR>
<INPUT TYPE="radio" NAME="rb1" VALUE="name"> Name<BR>
<INPUT TYPE="radio" NAME="rb1" VALUE="city"> City<BR>
<P><INPUT TYPE="submit" NAME="run" VALUE="Run Report"></DIR>
</FORM>
</HTML>
```

The FORM METHOD tag specifies that the /cgi-bin/myreport.sh CGI script is invoked when the Submit button is clicked. Adjust the URL of the CGI script to fit your particular environment.

In the INPUT tags, the TYPE=“radio” attribute defines a radio button. The VALUE attribute of the selected radio button is passed by the CGI script to the SQR program.

Creating the CGI Script

The CGI script is started when a user makes a request from a form. A CGI script can be any executable program. Do not call SQR directly as a CGI script—a PERL script, a shell script, or a C program all provide simpler routines for processing as a CGI script.

The CGI script:

1. Reads the contents of the standard input stream and parses them to obtain the values that were entered on the form.

   If the form has no input fields, this step is not required.

2. Identifies the output as being in HTML format by sending the *Content-type: text/html* string and an extra empty line to the standard output stream.

3. Invokes the SQR program.

   Values that the user entered on the form are passed to the SQR program by the CGI script and the command line.

4. Sends the generated .lis file to the standard output stream.

   The .htm file is not used because it points to the .lis file with a relative URL.

   The relative URL does not specify to the web browser where to find the .lis file. You should make provisions within your SQR program to send an error message.

The following Bourne shell is an example of a CGI script:

```
#! /bin/sh
# set the appropriate environment values
ORACLE_SID=oracle7; export ORACLE_SID
ORACLE_HOME=/usr2/oracle7; export ORACLE_HOME
SQRDIR=/usr2/sqr/bin; export SQRDIR
# identify the output as being HTML format
echo "Content-type: text/html"
echo ""
# get values from fill-out form using the POST method
read TEMPSTR
SORTBY=`echo $TEMPSTR | sed "s;.*rb1=;;
s;&.*;;"`
# invoke the SQR program
sqr7 /usr2/reports/myreport.sqr orauser/orapasswd \
    -PRINTER:ht -f/tmp/myreport$$.lis -I$SQRDIR "$SORTBY" \
     > /tmp/myreport$$.out 2>&1 < /dev/null
if [ $? -eq 0 ]; then
   # display the output
   cat /tmp/myreport$$.lis
else
   # error occurred, display the error
   echo "<HTML><BODY><PRE>"
   echo "FAILED TO RUN SQR PROGRAM"
   cat /tmp/myreport$$.out
   echo "</PRE></BODY></HTML>"
fi# remove temp files
rm /tmp/myreport$$.*
```

The script performs the following tasks:

1. Sets the necessary environment variables. Then it sends the Content-type: text/html string and an extra empty line to the standard output stream to identify the text as being HTML format.

2. Retrieves the value of the selected radio button into the *SORTBY* variable. The script passes the value to the SQR program on the command line.

3. Runs the SQR program. The script uses the /usr2/reports/myreport.sqr report file and generates the /tmp/myreport$$.lis file. In addition, the script redirects the standard input from /dev/null to prevent the program from stopping if the program requires any input. It also redirects the standard output to /tmp/myreport$$.out to capture any status messages. The $$ is the process ID of the program and is used as a unique identifier to prevent any multiuser problems.

4. Copies the generated report file to the standard output stream. If an error occurs, the script generates the status message file instead to enable the user to view the status messages. It then deletes any temporary files.

Passing Arguments to the SQR Program

You must modify the SQR program to accept values that the user enters on the form.

The following code example is the main procedure from sample program ex28b.sqr. It was modified to use the SORT BY value that is passed from the CGI script. The *$sortby* variable is obtained from the command line with an INPUT command and is used as dynamic variables in the ORDER BY clause. The modified lines are shown **like this:**

```
begin-procedure main
input $sortby 'Sort by' type=char
do html_on
do html_table('')
do html_tr('')
do html_th('')
print 'Name'   (3,1)
do html_th('')
print 'City'   (,32)
do html_th('')
print 'State'  (,49)
begin-select
  do html_tr('')
  do html_td('')
name  (,1,30)
  do html_td('')
city  (,+1,16)
  do html_td('')
state (,+1,5)
next-listing no-advance need=1
 let #grand_total = #grand_total + &tot
from customers
order by [$sortby]
end-select
```


Chapter 30

Generating Tagged PDF Output from SQR Program

Tagged PDF Overview

PDF is a file that contains a list of text, graphics, bookmarks, links, and other elements that make up an electronic document.

A PDF file follows a logical reading order that has images with descriptions, tagged tables with a structure, and tagged contents with headings, lists, and paragraphs, depending on the usage.

Note: Starting with SQR 8.53, SQR uses PDFLib 8.0.1 upgraded from PDFlib 3.0.3 library supplied by PDFLib GMBH to generate the tagged PDF documents by calling the respective Tagged PDF API calls.

Sample Program to Create Tagged PDF

Specifying Heading

The following program describes tagged content with heading:

```
begin_tag_heading
print 'SAMPLE HEADING' (+2,{C_MenuName})
end_tag_heading
```

SQR Program for using Paragraph in tagged content

The following program is an example of content tagging with paragraph:

```
begin_tag_paragraph
 print $test_val (+1,1)
 print $test_val1 (+1,1)
 print $test_val2 (+1,1)
 print $test_val3 (+1,1)
 print $test_val4 (+1,1)
 print $test_val5 (+1,1)
 end_tag_paragraph
```

Generating a Tagged Table in a PDF Report

Consider this sample program:

```
begin-procedure Report          ! Main report processing
uppercase $test
let #counter =#counter + 1
begin_tag_table

begin_table_tr
```

```
begin_tag_table_head
print 'TableHeader1'                    (#counter,{C_MenuName})
end_tag_table_head

begin_tag_table_head
print 'TableHeader2'                    (#counter,50)
end_tag_table_head

end_table_tr
begin-SELECT
MENUNAME &menuname

   begin_table_tr

   move &MenuName to $Field
   let #counter =#counter + 1
   begin_table_td
   print 'TESTDATA1'          (#counter,{C_MenuName})
   end_table_td

   begin_table_td
   print 'TESTDATA2'          (,50)
   end_table_td

   end_table_tr
FROM PSMENUDEFN
WHERE PSMENUDEFN.MENUNAME like 'A%'
ORDER BY PSMENUDEFN.MENUNAME
end-SELECT
end_tag_table
end-procedure
```

Generating a Tagged List in a PDF Report

The following example describes a Tagged List in a PDF repor:.

```
begin_tag_list
   begin_tag_list_index

   begin_tag_list_label
   print $Header4a_lbl (12, {colII})  Bold
   end_tag_list_label
   begin_tag_list_lbody
   print $Header4a_lbody (12, {colII_LBody})  Bold
   end_tag_list_lbody
   end_tag_list_index

   !Print $Header5a (13,{colII})   Bold
   begin_tag_list_index
   begin_tag_list_label
   Print $Header5a_lbl (13,{colII})   Bold
   end_tag_list_label
   begin_tag_list_lbody
   Print $Header5a_lbody (13,{colII_LBody})   Bold
   end_tag_list_lbody
   end_tag_list_index

   begin_tag_list_index
   begin_tag_list_label
   Print $Header6a_lbl (14,{colII})     Bold
   end_tag_list_label
   begin_tag_list_lbody
   Print $Header6a_lbody (14,{colII_LBody})     Bold
   end_tag_list_lbody
   end_tag_list_index

   begin_tag_list_index
   begin_tag_list_label
```



```
   Print $Header6b_lbl  (15, {colII})   Bold
   end_tag_list_label
   begin_tag_list_lbody
   Print $Header6b_lbody  (15, {colII_LBody})   Bold
   end_tag_list_lbody
   end_tag_list_index

   begin_tag_list_index
   begin_tag_list_label
   Print $Header8a_lbl  (16,{colII})    Bold
   end_tag_list_label
   begin_tag_list_lbody
   Print $Header8a_lbody  (16,{colII_LBody})    Bold
   end_tag_list_lbody
   end_tag_list_index
```

Generating Alternate Text for a Figure

The following is an example for generating alternate text for a figure:

```
 Begin-Program
  Let #Image_Length = 6
  Let #Image_Height = 8
  Print 'Look at this flower: ' (2,2)
  begin_tag_alt_text_figure ‘Fig:Rose-flower’
 Print-Image Flower (+2,5)
 Image-Size=(#Image_Length,#Image_Height)
Let #Curr_Line_Adj = #Image_Height +3
end_tag_alt_text_figure
 Print 'Isn''t it lovely?' (+#Curr_Line_Adj,2)
 End-Program
```

Tagged PDF in PeopleSoft Application

Enable Option of Tagged PDF in PIA will allow SQR engine to recognize the predefined Tagged PDF SQR commands.

By default, all SQR programs are in “not Tagged PDF” mode and will produce the normal PDF report.

To enable a tagged PDF option in PIA:

1. Navigate to Peopletools > Process Scheduler > Processes.

   For example, search for QR report GPINPS01.

2. Open the process name GPINPS01 and click the Override options tab.

3. Add the flag –PDF_TAG to the SQR Report. This flag should be added in a similar way as adding –TB,-S flags for an SQR Report.

Using Accessibility Checkers

Accessibility Checkers available in market can be used to check the tagged pdf report.

Tools that are available in market to check the accessibility of PDF documents are:

- PDF accessibility checker

- Adobe Acrobat Pro

Adobe Acrobat Pro can do a Quick full check to identify the accessibility compliance of a PDF report.

Chapter 31

Generating XML Output from SQR Program

Generating XML Output

An SQR program that generates XML output contains XML tags, which describe the data. You use the XML commands to generate XML output.

To generate XML output from an SQR program, you must:

1. Define the structure or template of the XML to be generated. Use DEFINE-XML-TEMPLATE command to define the template.

   You also can create nested XML by adding one template to another. The definition of the child template should precede that of the parent template.

2. Create XML-RECORD to define the XML template. To create XML-RECORD, you must specify the data. Each template will have an XML-RECORD to add data. This data is created in memory before being written to a disk.

   If XML templates are nested, you must add child XML-RECORDs before you add the parent XML-RECORD. Multiple child records can be added to a parent record by repeatedly adding child XML-RECORDs.

   Note: Ensure that child XML-RECORDs are not duplicated..

3. Write the XML records to a disk and use WRITE_TO_XML_FILE to write the XML records to an XML output file.

   Note: The system clears the parent template automatically after the XML records are written, but you must clear the child templates manually.

Note: XML output format is not applicable for any of the existing reports that are shipped. Only new SQRs or ones modified with the new commands will generate an XML file.

SQR Commands to Generate XML Output

The following table describes the SQR commands to generate XML output:

SQR Command	*Description*
DEFINE_XML_TEMPLATE	Defines the structure of the XML.
ADD_DATA_TO_XML_RECORD	Adds data to XML template to create an XML-RECORD before writing the XML file.

SQR Command	*Description*
ADD_CHILD_TO_XML_RECORD	Adds a child record to the XML template to create an XML-RECORD in a tree structure before it is pushed to the XML file.
ADD_ATTRIBUTE_TO_XML_RECORD	Adds an attribute to any record or element that has data.
WRITE_TO_XML_FILE	Writes the XML records to the XML output file.
(Optional) WRITE_CDATA_TO_XML_FILE (OPTIONAL)	Adds character data or CDATA to XML files that are not parsed. A CDATA section starts with *"<![CDATA["* and ends with *"]]>"*.
(Optional) EDIT_XML_ROOT	Modifies the root tag name.
(Optional) ADD_COMMENT_TO_XML_FILE	Adds comment as title to the XML file.
(Optional) CLEAR-XML-RECORD	Clears parent and child XML records.

Sample Program to Generate XML Output

The following program illustrates the use of XML commands to generate XML output:

```
let $pop = '123'
let #num = 100.9999999

EDIT_XML_ROOT 'Progressreport' 'default:student,n1:space1,n2:space2'

DEFINE_XML_TEMPLATE 'grand_child' 'default'
ADD_ELEMENT 'Column1'
ADD_ELEMENT 'Column2'
END_XML_TEMPLATE


DEFINE_XML_TEMPLATE 'child' 'default'
ADD_ELEMENT 'Column11'
ADD_ELEMENT 'Column22'
ADD_AS_CHILD 'grand_child'
END_XML_TEMPLATE

ADD_DATA_TO_XML_RECORD  'grand_child'  'Column1' 'ABC'
ADD_DATA_TO_XML_RECORD  'grand_child'  'Column2' '12'

ADD_DATA_TO_XML_RECORD  'child'  'Column11' $pop
ADD_DATA_TO_XML_RECORD  'child'  'Column22' #num
ADD_CHILD_TO_XML_RECORD 'child' 'grand_child'

WRITE_TO_XML_FILE 'child' 'default'

DEFINE_XML_TEMPLATE  'n1:subject' 'default'
ADD_ELEMENT 'subject_name'
ADD_ELEMENT 'subject_score'
END_XML_TEMPLATE

DEFINE_XML_TEMPLATE  'n2:subject' 'default'
ADD_ELEMENT 'subject_name2'
ADD_ELEMENT 'subject_score2'
END_XML_TEMPLATE

DEFINE_XML_TEMPLATE 'Address'   'default'
```

```
ADD_ELEMENT 'Building'
ADD_ELEMENT 'Street'
ADD_ELEMENT 'ZIP'
ADD_ELEMENT_FORCE_EMPTY  'Empty_column'
ADD_ELEMENT 'Non_Empty'
END_XML_TEMPLATE

DEFINE_XML_TEMPLATE 'student'   'default'
ADD_ELEMENT 'Name'
ADD_ELEMENT 'Age'
ADD_ELEMENT 'Date'
INHERIT_ELEMENTS 'Address'
ADD_AS_CHILD 'n1:subject'
ADD_AS_CHILD 'n2:subject'
END_XML_TEMPLATE

ADD_DATA_TO_XML_RECORD   'student'  'Name' 'ABC&'
ADD_DATA_TO_XML_RECORD   'student'  'Age'  12
ADD_DATA_TO_XML_RECORD   'student'  'Date' '1/1/2013'
ADD_DATA_TO_XML_RECORD   'student'  'Building' '123/4D'
ADD_DATA_TO_XML_RECORD   'student'  'Street' 'Baker Street'
ADD_DATA_TO_XML_RECORD   'student'  'ZIP' 654687

ADD_DATA_TO_XML_RECORD   'n1:subject'  'subject_name' 'Mathematics'
ADD_DATA_TO_XML_RECORD   'n1:subject'  'subject_score' '90'

ADD_CHILD_TO_XML_RECORD   'student'  'n1:subject'

CLEAR_XML_RECORD 'n1:subject'

ADD_DATA_TO_XML_RECORD   'n2:subject'  'subject_name2' 'Science'
ADD_DATA_TO_XML_RECORD   'n2:subject'  'subject_score2' 45

ADD_ATTRIBUTE_TO_XML_RECORD 'n2:subject' 'subject_score2' 'out_of' '100'

ADD_CHILD_TO_XML_RECORD   'student'  'n2:subject'

ADD_COMMENT_TO_XML_FILE  'Results of students'
WRITE_TO_XML_FILE 'student'  'default'
WRITE_CDATA_TO_XML_FILE  'Test CDATA'

DEFINE_XML_TEMPLATE 'MESSAGE_CATALOG' 'default'
ADD_ELEMENT 'MESSAGE_SET_NBR'
ADD_ELEMENT 'MESSAGE_NBR'
ADD_ELEMENT 'LANGUAGE_CD'
ADD_ELEMENT 'MESSAGE_TEXT'
END_XML_TEMPLATE

begin-select
MESSAGE_SET_NBR   &MESSAGE_SET_NBR
MESSAGE_NBR    &MESSAGE_NBR
LANGUAGE_CD    &LANGUAGE_CD
MESSAGE_TEXT   &MESSAGE_TEXT

   ADD_DATA_TO_XML_RECORD  'MESSAGE_CATALOG'  'MESSAGE_SET_NBR'  &MESSAGE_SET_NBR
   ADD_DATA_TO_XML_RECORD  'MESSAGE_CATALOG'  'MESSAGE_NBR'      &MESSAGE_NBR
   ADD_DATA_TO_XML_RECORD  'MESSAGE_CATALOG'   'LANGUAGE_CD'      &LANGUAGE_CD
   ADD_DATA_TO_XML_RECORD  'MESSAGE_CATALOG'  'MESSAGE_TEXT'    &MESSAGE_TEXT
   ADD_ATTRIBUTE_TO_XML_RECORD 'MESSAGE_CATALOG'  'MESSAGE_TEXT' 'LANGUAGE_CD' &LAN⇒
GUAGE_CD
   ADD_ATTRIBUTE_TO_XML_RECORD 'MESSAGE_CATALOG'  'MESSAGE_TEXT' 'MESSAGE_NBR' &MES⇒
SAGE_NBR
   WRITE_TO_XML_FILE 'MESSAGE_CATALOG'  'default'
   CLEAR_XML_RECORD 'MESSAGE_CATALOG'
from psmsgcatlang
where MESSAGE_SET_NBR = 92
end-select
```


The following is the sample XML output that is generated:

```
<?xml version="1.0"  encoding="UTF-8"?>
<Progressreport xmlns="student" xmlns:n1="space1" xmlns:n2="space2">
<child>
<Column11>123</Column11>
<Column22>100.999999</Column22>
<grand_child>
<Column1>ABC</Column1>
<Column2>12</Column2>
</grand_child>
</child>
<!--Results of students-->
<student>
<Name>ABC&amp;amp;</Name>
<Age>12</Age>
<Date>1/1/2013</Date>
<Building>123/4D</Building>
<Street>Baker Street</Street>
<ZIP>654687</ZIP>
<Empty_column></Empty_column>
<n1:subject>
<subject_name>Mathematics</subject_name>
<subject_score>90</subject_score>
</n1:subject>
<n2:subject>
<subject_name2>Science</subject_name2>
<subject_score2 out_of="100">45</subject_score2>
</n2:subject>
</student>
<![CDATA[
Test CDATA
]]>
<MESSAGE_CATALOG>
<MESSAGE_SET_NBR>92</MESSAGE_SET_NBR>
<MESSAGE_NBR>1</MESSAGE_NBR>
<LANGUAGE_CD>ARA</LANGUAGE_CD>
<MESSAGE_TEXT LANGUAGE_CD="ARA" MESSAGE_NBR="1">شجرة API</MESSAGE_TEXT>
</MESSAGE_CATALOG>
<MESSAGE_CATALOG>
<MESSAGE_SET_NBR>92</MESSAGE_SET_NBR>
<MESSAGE_NBR>1</MESSAGE_NBR>
<LANGUAGE_CD>ESP</LANGUAGE_CD>
<MESSAGE_TEXT LANGUAGE_CD="ESP" MESSAGE_NBR="1">ⓅTree  APIⓍ</MESSAGE_TEXT>
</MESSAGE_CATALOG>
<MESSAGE_CATALOG>
<MESSAGE_SET_NBR>92</MESSAGE_SET_NBR>
<MESSAGE_NBR>1</MESSAGE_NBR>
<LANGUAGE_CD>FRA</LANGUAGE_CD>
<MESSAGE_TEXT LANGUAGE_CD="FRA" MESSAGE_NBR="1">API arbre</MESSAGE_TEXT>
</MESSAGE_CATALOG>
<MESSAGE_CATALOG>
<MESSAGE_SET_NBR>92</MESSAGE_SET_NBR>
<MESSAGE_NBR>1</MESSAGE_NBR>
<LANGUAGE_CD>JPN</LANGUAGE_CD>
<MESSAGE_TEXT LANGUAGE_CD="JPN" MESSAGE_NBR="1">ﾂﾘｰ API</MESSAGE_TEXT>
</MESSAGE_CATALOG>
<MESSAGE_CATALOG>
<MESSAGE_SET_NBR>92</MESSAGE_SET_NBR>
<MESSAGE_NBR>3260</MESSAGE_NBR>
<LANGUAGE_CD>JPN</LANGUAGE_CD>
<MESSAGE_TEXT LANGUAGE_CD="JPN" MESSAGE_NBR="3260">ﾚﾍﾞﾙを切り替えられませんでした。</MESSAGE_TEXT⇒
>
</MESSAGE_CATALOG>
</Progressreport>
```

Generating XML Output in PeopleSoft Applications

You must schedule a process to generate XML output in PIA.

For more information about scheduling processes, see "Scheduling Process Requests" (PeopleTools 8.58: Process Scheduler).

To generate XML output in PIA:

1. Navigate to PeopleTools >Process Scheduler >System Process Request.
2. Click Search and select a run control ID.
3. Click Run.
4. Select XML as the output format.
5. Select the output type.
6. Click OK.

Chapter 32

Creating a Table of Contents

Using the DECLARE-TOC Command

Use DECLARE-TOC to define a table of contents and its attributes. When generating multiple reports and tables of contents from one SQR program, you can also use the TOC argument of the DECLARE-REPORT command.

You must issue the DECLARE-TOC command in the SETUP section of the program. For example:

```
begin-setup
      declare-toc toc_name
          for-reports = (all)
          dot-leader = yes
          indentation = 2
      end-declare
     .
     .
     .
end-setup
```

Following the DECLARE-TOC command, specify a table of contents name. Use the FOR-REPORTS argument to specify the reports within the SQR program that use this table of contents. Use (all) if you want all of the reports to use one table of contents. You need to specify individual report names only if you are generating multiple reports with different tables of contents from one program. Use DOT-LEADER to specify whether a dot leader precedes the page number. The default setting is NO, and the dot leader is suppressed in all HTML output except when you also specify -BURST:T with -PRINTER:HT. Use INDENTATION to specify the number of spaces by which each level is indented. (The default setting is 4.)

DECLARE-TOC also supports procedures that are frequently used for setup and initialization purposes, as described in this table:

Procedure	***Usage***
BEFORE-TOC	Specifies a procedure to be run before the table of contents is generated. If no table of contents is generated, the procedure does not run.
AFTER-TOC	Specifies a procedure to be run after the table of contents is generated. If no table of contents is generated, the procedure does not run.
BEFORE-PAGE	Specifies a procedure to be run at the start of each page.
AFTER-PAGE	Specifies a procedure to be run at the end of each page.

Using the TOC-ENTRY Command

Use TOC-ENTRY to place an entry into the table of contents and take the mandatory TEXT argument, which specifies the text to be placed in the table of contents. Legal text includes text literals, variables, and columns. To include levels in a table of contents, use the LEVEL argument, which specifies the level at which to place the text. If you do not specify this argument, the value of the previous level is used.

If you are writing programs that generate multiple reports, you can:

- Use the FOR-REPORTS argument of the DECLARE-TOC command to identify the reports to which the DECLARE-TOC command applies.
- Use the TOC argument of the DECLARE-REPORT command to specify the name of the table of contents for the report.

A program can have multiple DECLARE-TOC statements and multiple DECLARE-REPORT statements. However, you must include the FOR-TOCS argument in the DECLARE-TOC statements or the TOC argument in the DECLARE-REPORT statements.

To specify the name of the table of contents for a given report by using the TOC argument of the DECLARE-REPORT command, include code in the SETUP section of the program. For example:

```
 begin-setup
      declare-report
          toc = toc_name
      end-declare
     .
     .
     .
end-setup
```

Earlier, we modified the sample program ex7a.sqr to use the DECLARE-TOC and TOC-ENTRY commands. Then, we generated HTML output from the modified program by using the -PRINTER:EH and -PRINTER:HT command-line flags. In HTML, the table of contents file is a linked point of navigation for the online report.

However, you may also want to generate output files for printing hard-copy reports. The table of contents features can also perform this task. To test this assertion, run the modified version of the sample program ex7a.sqr and print it from an .lis file (or use -PRINTER:WP in Microsoft Windows). The table of contents output contains the traditional dot leaders and necessary page numbers relating to a hard-copy report.

See Using the DECLARE-PRINTER Command and Understanding the Sample Program for Printing Mailing Labels.

Adding a Table of Contents to the CUST.SQR Sample Program

The following program is based on cust.sqr, which is located in the SAMPLE (or SAMPLEW) directory. The program identifies the table of contents with the specific name of cust_toc. The dot leader is turned on. Indentation is set to 3. One table of contents level is set by using the LEVEL=1 argument or the TOC-ENTRY command. The BEFORE-PAGE and AFTER-TOC arguments of the DECLARE-TOC command are used to print simple messages here.

Table of Contents Sample Program 1

Consider this sample program:

```
begin-setup
  declare-toc cust_toc
    for-reports=(all)
    dot-leader=yes
    indentation=3
    after-toc=after_toc
    before-page=before_page
  end-declare
end-setup
begin-program
  do main
end-program
begin-procedure after_toc
  position (+1,1)
  print 'After TOC' () bold
  position (+1,1)
end-procedure
begin-procedure before_page
  position (+1,1)
  print 'Before Page' () bold
  position (+1,1)
end-procedure
begin-procedure main
begin-select
  print 'Customer Info' ()
  print '-'        (+1,1,62) Fill
name     (+1,1,25)
  toc-entry text = &name level = 1
cust_num (,35,30)
city     (+1,1,16)
state    (,17,2)
phone    (+1,1,15) edit (xxx)bxxx-xxxx
  position (+2,1)
from customers
order by name
end-select
end-procedure       ! main
begin-heading 3
 print $current-date (1,1) Edit 'DD-MON-YYYY'
 page-number (1,69) 'Page '
end-heading
```

Table of Contents Sample Program 2

The following program is also based on cust.sqr. It is similar to the previous program but declares two table of contents levels. This program also creates headings and footings that are specific to the table of contents. The FOR-TOCS argument of the BEGIN-HEADING and BEGIN-FOOTING commands enable you to specify, by name, the table of contents to which the particular heading or footing section applies. If the program is generating multiple reports with multiple tables of contents, then you can apply unique or common headings and footings to different reports and tables of contents. The table of contents heading of this program prints *Table of Contents* and the page number. The page numbers in the table of contents print as roman numerals. The table of contents footing prints *Company Confidential.*

```
begin-setup
  declare-report cust
  end-declare
  declare-toc cust_toc
    for-reports=(cust)
    dot-leader=yes
    indentation=3
    after-toc=after_toc
    before-page=before_page
  end-declare
```



```
  declare-variable
    integer #num_toc
    integer #num_page
  end-declare
end-setup
begin-program
  use-report cust
  do main
end-program
begin-procedure after_toc
  position (+1,1)
  print 'After TOC' () bold
  position (+1,1)
end-procedure
begin-procedure before_page
  position (+1,1)
  print 'Before Page' () bold
  position (+1,1)
end-procedure
begin-procedure main
begin-select
  print 'Customer Info' ()
  print '-'                (+1,1,62) Fill
name     (+1,1,25)
  toc-entry text = &name level = 1
cust_num (,35,30)
city     (+1,1,16)
state    (,17,2)
phone    (+1,1,15) edit (xxx)bxxx-xxxx
  position (+2,1)
  do orders(&cust_num)
  position (+2,1)
from customers
order by name
end-select
end-procedure ! main
begin-procedure orders (#cust_num)
  let #any = 0
begin-select
  if not #any
    print 'Orders Booked' (+2,10)
    print '-------------' (+1,10)
    let #any = 1
  end-if
b.order_num
b.product_code
order_date               (+1,10,20) Edit 'DD-MON-YYYY'
description              (,+1,20)
  toc-entry text = &description level=2c.price * b.quantity
(,+1,13) Edit $$$$,$$0.99
from  orders a, ordlines b, products c
where a.order_num = b.order_num
  and b.product_code = c.product_code
  and a.cust_num = #cust_num
order by b.order_num, b.product_code
end-select
end-procedure ! orders
begin-footing 3
 for-tocs=(cust_toc)
 print 'Company Confidential' (1,1,0) center
print $current-date (1,1) Edit 'DD-MON-YYYY'
end-footing
begin-heading 3
 for-tocs=(cust_toc)
 print 'Table of Contents' (1,1) bold center
 let $page = roman(#page-count)
 print 'Page ' (1,69)
 print $page ()
end-heading
begin-heading 3
 print $current-date (1,1) Edit 'DD-MON-YYYY'
```

```
 page-number (1,69) 'Page '
end-heading
```