



PeopleTools 8.12 Application Engine PeopleBook

PeopleTools 8.12 Application Engine PeopleBook

SKU MTAEr8SP1B 1200

PeopleBooks Contributors: Teams from PeopleSoft Product Documentation and Development.

Copyright © 2001 by PeopleSoft, Inc. All rights reserved.

Printed in the United States of America.

All material contained in this documentation is proprietary and confidential to PeopleSoft, Inc. and is protected by copyright laws. No part of this documentation may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, including, but not limited to, electronic, graphic, mechanical, photocopying, recording, or otherwise without the prior written permission of PeopleSoft, Inc.

This documentation is subject to change without notice, and PeopleSoft, Inc. does not warrant that the material contained in this documentation is free of errors. Any errors found in this document should be reported to PeopleSoft, Inc. in writing.

The copyrighted software that accompanies this documentation is licensed for use only in strict accordance with the applicable license agreement which should be read carefully as it governs the terms of use of the software and this documentation, including the disclosure thereof.

PeopleSoft, the PeopleSoft logo, PeopleTools, PS/nVision, PeopleCode, PeopleBooks, Vantive, and Vantive Enterprise are registered trademarks, and *PeopleTalk* and "People power the internet." are trademarks of PeopleSoft, Inc. All other company and product names may be trademarks of their respective owners.

Contents

About This PeopleBook

Before You Begin	ix
Related Documentation	x
Documentation on the Internet	x
Documentation on CD-ROM	x
Hardcopy Documentation	x
Typographical Conventions and Visual Cues	xi
Comments and Suggestions	xii

Chapter 1

Introducing Application Engine

What is Application Engine?	1-1
Why Use Application Engine?	1-2
Encapsulation	1-3
Upgrade Support	1-3
Reuse Business Logic	1-3
Built-In Restart Logic	1-4
Graphical User Interface	1-4
Enhanced SQL/Meta-SQL Support	1-5
Platform Flexibility	1-6
Data Dictionary Integration	1-7
Program Components and Terminology	1-7
Program	1-8
Sections	1-8
Steps	1-8
Actions	1-8
Application Engine State Record	1-10

Chapter 2

Application Engine Interface

Overview	2-1
Project Workspace	2-2
Object Workspace	2-3

Definition View	2-3
Program Flow View	2-5
Navigation	2-6
Switching Between Definition and Program Flow Views	2-7
Within the Program Flow View	2-7
Menus	2-8
Popup Menus.....	2-8
Definition View Popup Menu	2-8
Program Flow Popup Menu	2-9
Dropdown Menus.....	2-10
File Menu	2-10
Edit Menu.....	2-11
View Menu.....	2-12
Insert Menu	2-14
Toolbar Buttons	2-15
Designer Interface Tips.....	2-16
Viewing Definition Objects	2-16
Using the Plus and Minus Signs.....	2-16
Using the Folder Icons	2-16
Section Filtering.....	2-17
Activating Object Property Edit Boxes.....	2-19
Text Boxes	2-19
Drop-Down Lists.....	2-20
Toggles/Switches	2-20
Calendars.....	2-20
Selecting Objects.....	2-21
Definition Object Layout	2-21
Working with Definition Objects.....	2-22

Chapter 3

Creating Application Engine Definitions

Programs	3-1
Program Properties.....	3-2
General	3-3
State Records.....	3-3
Temp Tables.....	3-5
Advanced	3-7
Procedures Related to Programs	3-8
Sections.....	3-12
Section Properties.....	3-13

Filtering Sections	3-15
Execution Precedence	3-16
Inserting Sections	3-16
Locating Sections	3-18
Finding Object References (Call Sections)	3-19
Finding Sections within the current Program.....	3-20
Steps.....	3-20
Step Properties	3-20
Inserting Steps	3-22
Actions	3-24
Action Properties.....	3-24
Inserting Actions	3-26
SQL Actions.....	3-27
ReUse Statement	3-28
No Rows.....	3-29
Program Flow Actions	3-30
DO When	3-31
DO While	3-31
DO Select	3-31
DO Until.....	3-33
PeopleCode Actions	3-33
Call Section Actions.....	3-34
Section Name	3-35
Program ID.....	3-35
Dynamic	3-35
Program Properties.....	3-36
Sharing State Records	3-36
Log Message Actions	3-37
Action Execution Hierarchy.....	3-38

Chapter 4

Advanced Development

State Records	4-1
Overview	4-1
Sharing State Records	4-3
Choosing a Record Type for State Records	4-4
Application Engine Commit	4-4
%TruncateTable Considerations	4-5
"No Rows" Considerations	4-5
Re-Using Statements	4-6

ReUse (SQL Action Property)	4-6
Bulk Insert	4-7
Developing for Restart	4-8
Enabling Restart	4-8
Disabling Restart	4-9
How Restart Works	4-9
Deciding to Restart	4-10
At the Program Level	4-11
At the Section Level	4-12
At the Step Level	4-12
Abends	4-13
Using Temporary Tables	4-14
Overview	4-14
Application Designer: Creating Temporary Table Instances	4-16
Defining Temporary Tables	4-17
Building Temporary Tables	4-17
Application Engine: Managing Temporary Table Instances	4-18
Assigning Temporary Tables to Programs	4-18
Adjusting Meta-SQL	4-19
Making External Calls	4-21
PeopleTools Options: Specifying the Number of Temporary Tables	4-22
Temp Table Instances (Total)	4-22
Online vs. Batch Program Runs	4-23
Run Time Behavior	4-24
Overview	4-24
Table Locking	4-24
Sample Implementation	4-25
Application Engine Meta-SQL	4-26
Application Engine Meta-SQL	4-28
%Bind	4-28
%ExecuteEdits	4-31
%Select	4-32
%SelectInit	4-33
%SQL	4-34
%Table	4-35
%TruncateTable	4-36
%UpdateStats	4-36
Application Engine Macros	4-39
%ClearCursor	4-39
%Execute	4-40

%Next and %Previous.....	4-41
%RoundCurrency	4-41
Application Engine System (Meta) Variables.....	4-42
%AeProgram.....	4-42
%AeSection.....	4-42
%AeStep.....	4-43
%JobInstance	4-43
%ProcessInstance	4-43
%ReturnCode	4-43
%RunControl	4-43
%AsOfDate	4-43
%Comma.....	4-43
%LeftParen.....	4-43
%RightParen	4-43
%Space.....	4-43
%SQLRows.....	4-43
%List	4-44
Using PeopleCode in Application Engine Programs	4-44
Application Engine's Purpose	4-45
Environment Considerations	4-45
Passing Parameters through CallAppEngine.....	4-46
Defining Global Variables	4-46
State Records.....	4-47
IF, THEN Logic	4-47
Variable Scope	4-48
Action Execution Hierarchy.....	4-49
Using PeopleCode in Loops.....	4-50
AeSection Object.....	4-51
Synchronous Online Application Engine Calls (CallAppEngine)	4-51
Events to Trigger CallAppEngine.....	4-52
Process Instance	4-53
Save Event Details	4-53
FieldChange Event Details.....	4-54
File Layout Object.....	4-54
Calling COBOL Modules (RemoteCall).....	4-55
PTPECOBL.....	4-55
Syntax and Parameters	4-56
Commit.....	4-58
Restrictions.....	4-58
PeopleTools APIs.....	4-59

CommitWork.....	4-59
Notes on Various PeopleCode Objects and Functions	4-60
Do When	4-60
Dynamic SQL.....	4-60
Sequence Numbering	4-61
Rowsets	4-61
Math Functions	4-61
SQL Objects.....	4-62
Arrays.....	4-63
Set Processing.....	4-63
What is Set Processing?	4-64
Advantages of Set Processing	4-65
Using Set Processing Effectively	4-66
SQL Expertise	4-66
Planning	4-66
Temporary Tables	4-67
Tips.....	4-69
Examples of Set Processing	4-70
Payroll	4-71
Using Temporary Tables.....	4-71
Platform Issues	4-76
Dynamic SQL	4-77
Debugging Application Engine Programs	4-78
Enabling the Application Engine Debugger.....	4-78
Debugging Options	4-81
Before You Get Started.....	4-81
Quit.....	4-82
Exit	4-82
Commit.....	4-82
Break	4-82
Look	4-84
Modify.....	4-85
Watch	4-86
Step over	4-87
Step Into	4-88
Step Out of	4-88
Go.....	4-89
Run to commit.....	4-89

Chapter 5

Administration

Application Engine Administration Interface	5-1
Invoking Application Engine Programs	5-2
Batch Server Considerations	5-3
Supported Execution Platforms.....	5-3
Abnormal Program Ends.....	5-3
Tuxedo Requirements	5-3
TOOLBINSRV Parameter	5-3
Process Scheduler.....	5-3
Application Engine Process Request Page	5-5
Overview	5-5
Page Controls	5-6
Upgrading Request Pages from PeopleTools 7.5	5-7
PeopleCode—CallAppEngine.....	5-8
Command Line.....	5-9
Application Engine Server Caching	5-12
CacheBaseDir.....	5-12
Server Caching.....	5-13
Tracing Application Engine Programs	5-13
Enabling Application Engine Traces.....	5-14
Command Line.....	5-14
Server Configuration Files	5-15
Configuration Manager	5-16
Locating the Trace File	5-17
Understanding the Trace Results.....	5-17
Step.....	5-18
SQL	5-19
Application Engine Statement Timings	5-20
DB Optimizer Trace.....	5-31
Temporary Table Performance Considerations.....	5-36
Initial Estimates.....	5-36
Online Temporary Table Allocation	5-36
Viewing Temporary Table Usage.....	5-36
Managing Abends.....	5-37
Restarting Application Engine Programs.....	5-38
Restarting an Application Engine Program.....	5-38
Command Line.....	5-39
Process Request Page	5-40
Deciding to Start over or Restart.....	5-40
Troubleshooting	5-41

"Bad restart" Messages	5-41
The "Suspend" Error	5-41

Index

ABOUT THIS PEOPLEBOOK

In this book we discuss the use of Application Engine, a PeopleTool designed to help you develop, test, and run background SQL processing programs. Its chapters explain the concepts and advantages of Application Engine, how to create programs using the Application Engine Designer, how to run and debug programs, and the use of the special tools we provide to maintain your programs.

This book is written for technical users and programmers who will be running or developing applications using Application Engine. To take full advantage of the information covered in this book, we recommend that you have a basic understanding of how to use PeopleSoft applications. In other words, you should be familiar with how to navigate your way around the system and how to add, update, and delete information using PeopleSoft tables and panels. You should also be comfortable using Microsoft® Windows.

This document assumes that you are familiar with PeopleTools, specifically Application Designer, PeopleCode, Process Scheduler, and to some extent Mass Change. It also assumes a familiarity with structured programming languages, such as COBOL, relational database concepts, and SQL. Being a SQL expert is vital to programming efficient Application Engine programs.

The Application Engine is a powerful PeopleTool designed for developing efficient, high-performance, batch programs. Developers who use Application Engine should have experience designing and developing high-volume batch programs using COBOL or SQR.

The following links take you to the beginning of each chapter within this PeopleBook.

Introducing Application Engine This chapter provides an introduction to Application Engine including, why to use it, what are its benefits, terminology, and the components.

Application Engine Interface This chapter describes the interface you use within Application Designer to create your Application Engine programs.

Creating Application Engine Definitions This chapter covers the fundamentals of creating an Application Program.

Advanced Development This chapter covers more detailed topics related to Application Engine programs such as meta-SQL, set processing, and using PeopleCode.

Administration This chapter covers topics that system administrators will be interested in such as restarting Application Engine and improving performance.

Before You Begin

To benefit fully from the information covered in this book, you need to have a basic understanding of how to use PeopleSoft applications. We recommend that you complete at least one PeopleSoft introductory training course.

You should be familiar with navigating around the system and adding, updating, and deleting information using PeopleSoft windows, menus, and pages. You should also be comfortable using the World Wide Web and the Microsoft® Windows or Windows NT graphical user interface.

Related Documentation

To add to your knowledge of PeopleSoft applications and tools, you may want to refer to the documentation of the specific PeopleSoft applications your company uses. You can access additional documentation for this release from PeopleSoft Customer Connection (www.peoplesoft.com). We post updates and other items on Customer Connection, as well. In addition, documentation for this release is available on CD-ROM and in hard copy.



Important! Before upgrading, it is *imperative* that you check PeopleSoft Customer Connection for updates to the upgrade instructions. We continually post updates as we refine the upgrade process.

Documentation on the Internet

You can order printed, bound versions of the complete PeopleSoft documentation delivered on your PeopleBooks CD-ROM. You can order additional copies of the PeopleBooks CDs through the Documentation section of the PeopleSoft Customer Connection Web site: <http://www.peoplesoft.com/>

You'll also find updates to the documentation for this and previous releases on Customer Connection. Through the Documentation section of Customer Connection, you can download files to add to your PeopleBook library. You'll find a variety of useful and timely materials, including updates to the full PeopleSoft documentation delivered on your PeopleBooks CD.

Documentation on CD-ROM

Complete documentation for this PeopleTools release is provided in HTML format on the PeopleTools PeopleBooks CD-ROM. The documentation for the PeopleSoft applications you have purchased appears on a separate PeopleBooks CD for the product line.

Hardcopy Documentation

To order printed, bound volumes of the complete PeopleSoft documentation delivered on your PeopleBooks CD-ROM, visit the PeopleSoft Press Web site from the Documentation section of PeopleSoft Customer Connection. The PeopleSoft Press Web site is a joint venture between PeopleSoft and Consolidated Publications Incorporated (CPI), our book print vendor.

We make printed documentation for each major release available shortly after the software is first shipped. Customers and partners can order printed PeopleSoft documentation using any of the following methods:

Internet

From the main PeopleSoft Internet site, go to the Documentation section of Customer Connection. You can find order information under the Ordering PeopleBooks topic. Use a Customer Connection ID, credit card, or purchase order to place your order.

PeopleSoft Internet site: <http://www.peoplesoft.com/>.

Telephone

Contact Consolidated Publishing Incorporated (CPI) at **800 888 3559**.

Email

Email CPI at callcenter@conpub.com.

Typographical Conventions and Visual Cues

To help you locate and interpret information, we use a number of standard conventions in our online documentation.

Please take a moment to review the following typographical cues:

`monospace font`

Indicates PeopleCode.

Bold

Indicates field names and other page elements, such as buttons and group box labels, when these elements are documented below the page on which they appear. When we refer to these elements elsewhere in the documentation, we set them in Normal style (not in bold).

We also use boldface when we refer to navigational paths, menu names, or process actions (such as **Save** and **Run**).

Italics

Indicates a PeopleSoft or other book-length publication. We also use italics for *emphasis* and to indicate specific field values. When we cite a field value under the page on which it appears, we use this style: ***field value***.

We also use italics when we refer to words as words or letters as letters, as in the following: Enter the number *0*, not the letter *O*.

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 W.

Jump links	Indicates a jump (also called a link, hyperlink, or hypertext link). Click a jump to move to the jump destination or referenced section.
Cross-references	The phrase For more information indicates where you can find additional documentation on the topic at hand. We include the navigational path to the referenced topic, separated by colons (:). Capitalized titles in <i>italics</i> indicate the title of a PeopleBook; capitalized titles in normal font refer to sections and specific topics within the PeopleBook. Cross-references typically begin with a jump link. Here's an example:

For more information, see Documentation on CD-ROM in *About These PeopleBooks*: Related Documentation.

• Topic list	Contains jump links to all the topics in the section. Note that these correspond to the heading levels you'll find in the Contents window.
--------------	--------------------------------------------------------------------------------------------------------------------------------------------



Name of Page or
Dialog Box

Opens a pop-up window that contains the named page or dialog box. Click the icon to display the image. Some screen shots may also appear inline (directly in the text).



Text in this bar indicates information that you should pay particular attention to as you work with your PeopleSoft system. If the note is preceded by **Important!**, the note is crucial and includes information that concerns what you need to do for the system to function properly.



Text in this bar indicates For more information cross-references to related or additional information.



Text within this bar indicates a crucial configuration consideration. Pay very close attention to these warning messages.

Comments and Suggestions

Your comments are important to us. We encourage you to tell us what you like, or what you would like changed about our documentation, PeopleBooks, and other PeopleSoft reference and training materials. Please send your suggestions to:

PeopleTools Product Documentation Manager
PeopleSoft, Inc.
4460 Hacienda Drive
Pleasanton, CA 94588

Or send comments by email to the authors of the PeopleSoft documentation at:

DOC@PEOPLESOFT.COM

While we cannot guarantee to answer every email message, we will pay careful attention to your comments and suggestions. We are always improving our product communications for you.

CHAPTER 1

Introducing Application Engine

Application Engine is the PeopleTool that you use to develop batch or online programs that are effective alternatives to COBOL or SQR programs (for non-report processing). Using Application Engine, you can create programs that perform high-volume, background processing against your data.

Our in-house testing reveals that a well-constructed and properly tuned Application Engine program achieves performance metrics comparable to COBOL and SQR. In some cases, the Application Engine program can out-perform both COBOL and SQR. However, as you'll see later in this chapter, there are many other reasons to choose Application Engine to develop your batch programs besides performance.

To invoke an Application Engine program from a PeopleSoft application, you can use Process Scheduler or you can use PeopleCode. You can also invoke an Application Engine program manually from the command line. Most Application Engine programs are scheduled to run using the Process Scheduler.

After reading this section, you will be familiar the following:

- The components of Application Engine.
- What Application Engine is designed to do.
- How SQL and PeopleCode fit in.
- The benefits of Application Engine over SQR or COBOL.
- The basic components and terminology with which you will need to become familiar before you delve further into this PeopleBook.

What is Application Engine?

Application Engine is comprised of two distinct components: the "front-end" where you define your batch program and the "back-end," which entails running and monitoring your program. This is analogous to other PeopleTools, such as Process Scheduler, where a definitional component is separate from the run-time component.

You can access the Application Engine front-end, or development component, by way of Application Designer. This is where you initially define and, later, maintain an Application Engine program. The back-end is the PeopleTools executable that runs the Application Engine program you developed.

In the realm of Application Engine, a *program* is a set of SQL statements, PeopleCode, and Program Control Actions (that allow looping and conditional logic) defined in Application

Designer that performs a business process. You can use Application Engine for straight, row-by-row processing, but the most efficient Application Engine programs are written to perform set-based processing.



For more information on set-based processing, see Set Processing.

Application Engine does not generate SQL or PeopleCode. It is only designed to execute the SQL and PeopleCode that you include in an Application Engine Action as part of your overall program. Think of Application Engine as a tool that allows you to define a program's framework, as in its algorithm, structure, and looping constructs. But, Application Engine relies on the developer to include the "guts" within the program's framework, such as the SQL and PeopleCode. As you'll see later in this section, there are many reasons why you should choose Application Engine as the "shell" to execute your business logic.

We assume that you have the expertise necessary to construct efficient SQL and PeopleCode to apply your business rules. Application Engine is arguably one of the most powerful tools that PeopleTools offers, but it can not compensate for inefficient program logic. As with any software development tool, it can only do what you "tell" it to do.

As far as SQL is concerned, you can use your native SQL query utility to compose your SQL statements, or you can just use the PeopleTools SQL Editor. Where you decide to write your SQL is irrelevant, although most will find that using the SQL Editor is more convenient. The Application Engine designer interface automatically launches the SQL Editor where appropriate. If you do opt to compose your SQL outside of the SQL Editor, when you're done all you have to do is paste it into the appropriate Action within your Application Engine program.

Being able to add PeopleCode to your Application Engine programs can improve performance, and using PeopleCode means that you don't have to use SQL for everything, such as assignments, conditions, and SQL text building. Also, it allows you to take advantage of features such as the PeopleCode File objects and Interlink objects, Component Interfaces, Application Messaging, and so on. PeopleCode also allows you to reuse online functions and control the flow of the Application Engine program.

To add PeopleCode, just use the PeopleCode Editor. When you've added a PeopleCode Action, you can automatically launch the PeopleCode Editor from the Application Designer interface.

Why Use Application Engine?

As stated previously, we designed Application Engine for batch processing where you have a large (or small) amount of data that needs to be processed without user intervention. An example of a typical batch process would be calculating the salaries in payroll processing (not printing the checks). Another example might be converting data (money) from one currency to another.

Well, anyone who is considering the use of Application Engine probably already knows of all the uses for a tool that helps you to develop batch programs. The real question you're probably asking is "Why should I use Application Engine instead of COBOL or SQR?" Hopefully, after reading the following sections, you'll see the advantages that Application Engine offers over its counterparts, COBOL and SQR.

Encapsulation

Unlike applications developed using COBOL or SQR, Application Engine programs reside *completely* within your database. With Application Engine, you do not have to compile your programs, there are no statements to store, and there is no need to directly interact with the operating environment you use.

You can develop your entire program from scratch, including the table definitions, all within the Application Designer, and then you can run and debug your applications without leaving the PeopleTools environment.

Upgrade Support

Since Application Engine program definitions are defined in and stored as Application Designer objects, they join the growing ranks of PeopleTools components that you can easily upgrade using Application Designer.

You can upgrade Application Engine definitions by project, program, or Section.

Also, you can insert Records and Application Engine programs in the same project. This means that if you change your Record definition, you can put the modified Record into a “project”, and you can then also include in the project any Application Engine programs affected by the change—along with affected panels, and so on. Then you upgrade the entire project all at once. With COBOL and SQR, you have to do the record change then modify your batch programs using some other mechanism all while make sure to keep track of what was changed and how and where it affects the batch program. With Application Engine and PeopleTools, it’s all taken care of automatically.



For more information on upgrading Application Engine programs, see your upgrade documentation.

Reuse Business Logic

Application Engine programs can now invoke PeopleCode. This means that from Application Engine you can call common PeopleCode functions that you use throughout your PeopleSoft system. Conversely, PeopleCode can now invoke an Application Engine program.

This allows many opportunities for sharing and reusing code. Regardless of where you’ve stored a piece of code, you can reuse it.



Note. Reusing business logic between batch and online generally applies to row-by-row processing, and not set processing.

Built-In Restart Logic

Within each Application Engine program, you must define how frequently your program will issue a COMMIT. After doing so, each COMMIT becomes a "checkpoint" that Application Engine uses to locate where within a program to restart after an abend.

The restarted Application Engine program locates the last checkpoint, or the last successfully executed and committed Step, and continues with the processing from that point. This type of *built-in* logic does not exist in COBOL or SQR. You need to code any restart logic for your programs yourself.

With COBOL or SQR, you not only have to resolve the issue that caused the error; you may also need to "undo" what the program processed before the failure. Otherwise, you may perform duplicate processing, which leads to inaccurate data.

Besides offering the obvious advantage of having built-in restart capability, Application Engine's restart capabilities also influence the design and performance of an Application Engine program. Having restart functionality allows the developer to commit more often, without the added burden of coding additional restart logic with every commit. This reduces the overall impact on end users and processes while the background program is running.



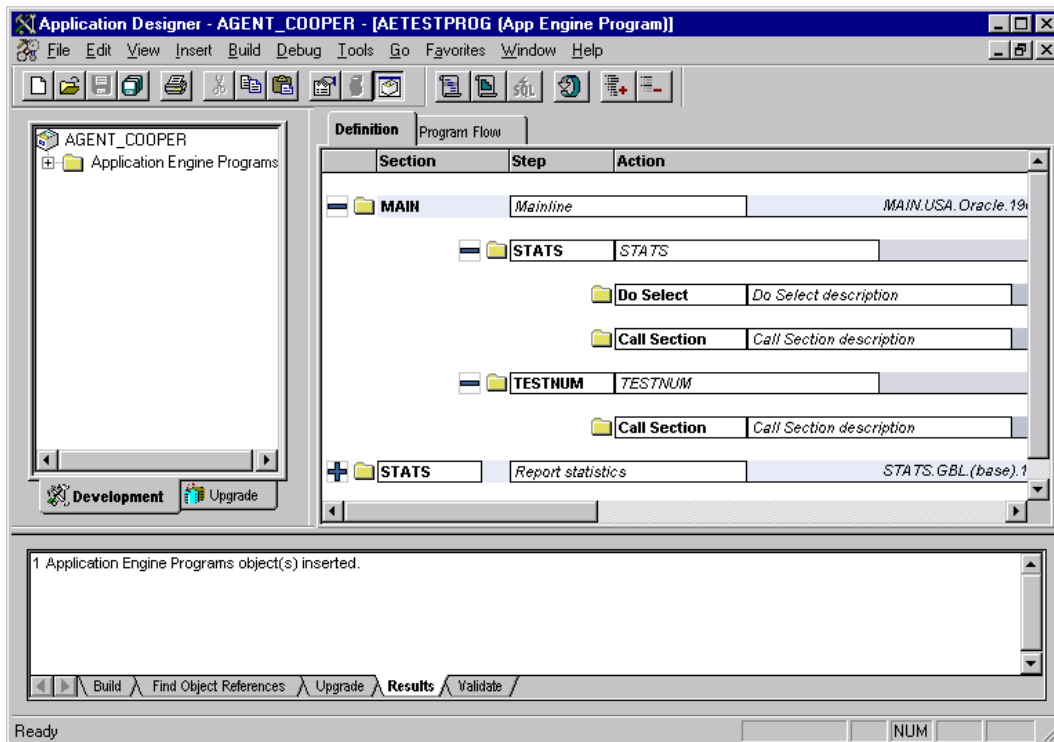
For more information see [Developing for Restart](#).

Graphical User Interface

Application Engine is fully integrated with the other design time PeopleTools. This means that you use the Application Designer, an intuitive, graphical interface, to define your Application Engine program.

The Application Designer offers Application Engine developers the following benefits:

- Ability to open multiple Application Engine programs concurrently.
- Ability to work on the entire Application Engine program at once. Users of previous versions might remember working on one Section at a time.
- Easy access to the PeopleCode and SQL editors.
- Two views of your program. The **Definition** view is where you create and modify your programs. The **Program Flow** view allows you to see the actual order in which your program will execute the statements.



Application Engine Program Definition in Application Designer



For more information see [Creating your Application Engine Programs](#).

Enhanced SQL/Meta-SQL Support

You can write your SQL within Application Engine, or you can copy SQL statements into Application Engine from any SQL utility with few, if any, changes. This enables you to write and tune your SQL statements before you try to incorporate them into an Application Engine program. In fact, you may find yourself spending more time developing in an external SQL utility, and then cutting and pasting your SQL into the corresponding Application Engine Actions.

RDBMS platforms have many differing syntax rules, especially in regard to date, time and other numeric calculations. For the most part, you can work around differing syntax using PeopleSoft meta-SQL, which Application Engine supports. This language is designed to replace RDBMS-specific SQL syntax with a standard syntax, called meta-strings.

For example, if you reference a date field called AE_DATE in a SELECT clause, you specify it as %DATEOUT(AE_DATE). This meta-string will be resolved to the appropriate value as required by the current RDBMS platform.



For more information on PeopleSoft meta-SQL, see [Application Engine Meta-SQL](#).

In addition, PeopleSoft meta-SQL enables you to dynamically generate portions of your SQL statements. For example, if you wanted to join two tables on their common keys, you could use the following code:

```
%Join(COMMON_KEYS, PSAESECTDEFN ABC, PSAESTEPDEFN XYZ )
```

At runtime, the function would be expanded into the following:

```
ABC.AE_APPLID = XYZ.AE_APPLID

AND ABC.AE_SECTION = XYZ.AE_SECTION

AND ABC.DBTYPE = XYZ.DBTYPE

AND ABC.EFFDT = XYZ.EFFDT
```



Note. Much of the recent PeopleSoft meta-SQL (constructs added since PeopleTools 8) is not supported for COBOL or SQR programs.

Platform Flexibility

Even with the use of meta-SQL, there may be instances when you have to write and execute different SQL in your Application Engine program, depending on the RDBMS platform on which you intend your program to run. In COBOL programs that use stored statements, this would mean writing a different script file, called a "delta", for each platform.

If using dynamic SQL, you'd have to control it with something similar to the following statement:

```
IF DBTYPE-SYBASE OF SQLRT...
```

However, in Application Engine, you have the ability to call different versions of a Section for different platforms. You can specify a Section to be one of the following:

- *(base)*. Think of this as the common denominator or global version.
- Any supported RDBMS platform, as in DB2/UNIX, Oracle, Informix, and so on.

Most Sections will only have a (base) version, and, even in cases where you refine a particular Section for a specific RDBMS, a (base) version of that Section will also exist. When Application Engine executes a Section, it will use the version that's specific to the current database platform if one exists. If there is no RDBMS-specific version, Application Engine uses the (base) version.

Within platform-specific Sections, you also have the ability to call base portions of SQL statements by using the %SQL construct. This means you can write your generic SQL portions just once, and then you can reference them from your different platform versions.

Even if you don't intend to develop platform specific versions, using %SQL is still a benefit since you can refer to external SQL objects from within Application Engine SQL. This allows for reusing SQL and dynamic SQL.



For more information on %SQL, see Application Engine Meta-SQL.

Data Dictionary Integration

Unless you've developed them to anticipate changes in field attributes, COBOL applications usually need to be modified when the definitions of your meta-data change. For instance, if a developer increases a field's length, then it may need to be changed in every instance where the COBOL program references this field as a BIND or SELECT variable.

Correcting such discrepancies can consume a considerable amount of time and resources, and, if not handled properly, an unaccounted for change in your meta-data can cause interesting errors that are often difficult to pinpoint. For example, if the length of a field in your COBOL code is wrong, one of the following *may* occur:

- An error message may appear.
- The field may get truncated.
- Or, the program may run without a problem.

If you've ever had to perform such a manual update of COBOL code, then there's no need to imagine the degree of irritation a programmer can experience. Regardless of the outcome, by using COBOL, you are potentially restricting other developers from making changes to the meta-data since a simple change may send a time consuming, ripple effect throughout the system.

On the other hand, with Application Engine, developers enjoy the freedom of changing meta-data as needed thanks to the PeopleTools data dictionary.

One of the cornerstones of PeopleSoft functionality is Application Designer. Because of the way it works, most field attributes, such as type, length, and scale, only need to be specified once, and that change will be reflected globally. If the same field appears on more than one record, you know that the same attributes reside in each record definition. If needed, there are attributes you can override at the record level, such as a default value or whether a field is a key and so on.

For the most part, these global definitions, our meta-data, allow a developer to change the database to meet the site's current needs and then ALTER the affected tables, as needed. Although some cosmetic changes to panels might be required in some cases, typically, no SQL or PeopleCode program changes are required.

Because Application Engine works in harmony within the PeopleSoft system and references the data dictionary for object definitions, changes to meta-data in the database have little or no impact on Application Engine programs.

Program Components and Terminology

An Application Engine Program is made up of several key components. It is important to understand the relationship between the components and the hierarchy in which they exist. Although you will encounter many more terms that are specific to Application Engine within this

book, here we cover the basic terms with which you will want to become familiar before you continue.

If you are an experienced Application Engine developer, you may already be familiar with these basic terms.

Program

An Application Engine program identifies the set of processes to execute a given task. A program must contain at least one Section. The execution of the program always starts with the Section defined as “MAIN.”

Sections

An Application Engine Section is comprised of one or more Steps and is equivalent to a COBOL paragraph or an SQR procedure. All Application Engine programs must contain at least one section entitled “MAIN.”

A Section is a set of ordered Steps that gets executed as part of a program. You can call Sections (and other programs) from Steps within other Sections.

Steps

A Step is the smallest unit of work that can be committed within a program. Although, you can use a Step to execute a PeopleCode command or log a message, typically, you'll use a Step to execute a SQL statement or to call another Section. The SQL or PeopleCode that a Step executes are the Actions within the Step.

When a Section gets called, its Steps execute sequentially. Every program begins by executing the first Step of the required Section called MAIN and ends after the last Step in the last Section completes successfully.

Actions

There are multiple types of Actions that you can specify to include within a Step. Keep in mind that it is common to have multiple Actions associated with a single Step. The following sections briefly describe each type of Action. These Actions are explained in more detail when we approach more advanced topics related to building Application Engine programs.

Program Flow Actions

A Program Flow Action contains a SQL SELECT statement designed to return results on which subsequent Actions depend. For instance, if a SELECT returns no rows, subsequent Actions may not need to execute. A Program Flow Action is equivalent to a COBOL PERFORM statement and has similar constructs.

Here are the four types of Program Flow Actions:

- Do While.

- Do When.
- Do Select.
- Do Until.

SQL

Most SQL Actions are comprised of a single SQL statement. These statements can perform the following types of SQL statements:

- UPDATE
- DELETE
- INSERT
- SELECT

The SQL Action differs from the Program Flow Actions, which also contain SQL, in that the SQL Action does not control the flow of the program.

PeopleCode

You can include PeopleCode in, of course, the PeopleCode Action. Application Engine PeopleCode provides an excellent way to build dynamic SQL, perform simple IF/ELSE edits, set defaults, and other operations that don't require a trip to the database. It also gives you the ability to reference and change active Application Engine State Records.

Most importantly, PeopleCode provides access to the PeopleSoft Internet Architecture's integration technologies such as Application Messaging, Business Interlinks, Component Interfaces, and XML File Processing.



For more information on using PeopleCode Actions, see PeopleCode Actions.

Log Message

A Log Message Action can be used to write a message to the MESSAGE_LOG based on a particular condition in your program. This gives your program multi-language capability. The system stores the message generically as a message set, message number, and parameter values. When the end user views the messages using the Application Engine Messages panel, PeopleTools retrieves the appropriate message string from the Message Catalog based on the user's language preference.

Call Section

You can also insert an Action that calls another Section. The “called” Section can be in the same program as the calling Section, or it can be in an *external* program. This allows you to “chunk” your program into more maintainable, reusable pieces. If a Section already exists in one program, rather than copying it into another program, just call it.



Note. Application Engine supports up to 99 levels of nested Call Section Actions. For example, the first called Section can call a second, which can call a third, and so on, and so on, up to 99 calls.

Application Engine State Record

The State Record is a PeopleSoft record that must be created and maintained by the Application Engine developer. This record defines the fields a program uses to pass values from one Action to another. Think of the fields of the Application Engine State Record as comprising the working storage for your Application Engine program.

An Application Engine State Record can be either a physical record or a work record, and any number of State Records can be associated with a program. Physical State Records must be keyed by process instance.



For more information on State Records, see State Records.

CHAPTER 2

Application Engine Interface

To create your Application Engine programs you use the Application Designer. This interface presents a graphical format for creating Application Engine program definitions just as you would record and page definitions. You'll find that all of the interface benefits that come with the Application Designer, such as printing, copying, opening files, and so on, will also apply to the Application Engine Designer. Having an Application Engine definition open on the desktop will activate the Application Engine-specific menus.

While the Application Designer interface is intuitive (and familiar to previous Peoplesoft developers), there are some concepts that need to be understood before you begin developing an Application Engine program. Even if you're an experienced Application Engine developer, you'll probably want to read these sections to acquaint yourself with the controls and tricks of the interface.

As you build programs, you'll rely on the tools and options discussed in this chapter. Most of the tools will seem familiar if you're comfortable with Windows, PeopleTools, and the elements of PeopleSoft pages. The rest will become familiar as you hone your Application Engine program development skills.

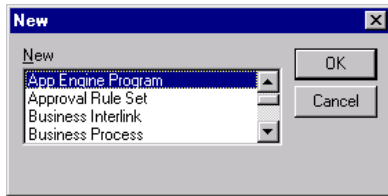
For the most part, you'll create and maintain Application Engine projects and definition objects in the same manner that you create and maintain other PeopleTools projects in the Application Designer.

After reading this section, you will be familiar with the following items:

- The Application Designer interface (as it relates to creating Application Engine definitions).
- Adding Sections, Steps, and Actions.
- The foundation of Application Engine Program Development.

Overview

You create, modify and review your Application Engine program using the Application Designer. You open the Application Engine Definition just as you would any other definition, such as Page, Record, Menu, and so on.



Creating a New Application Engine Definition

Because the Application Engine Definition is an Application Designer object, Application Engine Developers can take advantage of all the interface benefits that Application Designer offers, which include:

- Integration with other designer tools, such as the record designer and project/upgrade support..
- Fundamental interface functions, such as printing, copy and paste functionality, deleting, renaming, and context, or pop-up, menus.
- Change control support allows you to manage access to Application Engine objects by locking records. This prevents concurrent users from overwriting each other's updates.
- Multiple window support, which allows a developer to have multiple Application Engine programs open simultaneously. You can also have the PeopleCode and SQL Editor open if the program you're developing is open.
- You can easily launch the PeopleCode Editor and the SQL Editor from within the Application Engine Designer interface, as needed.

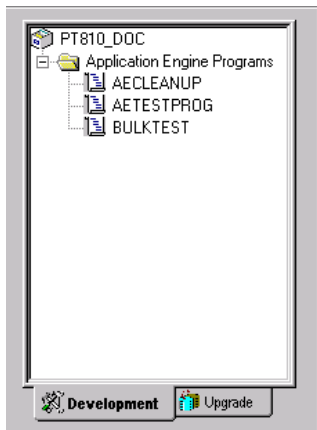


For more information on the Application Designer interface see Using the Application Designer.

Project Workspace

If you have the Development tab depressed for the Project Workspace you will see a standard Windows tree control that represents the currently loaded project definition.

The root of the tree represents the Application Engine project and the Application ID is the top-level node. Beneath that are the individual Application Engine programs that make up your project. Projects enable you to easily manage sets of related Application Engine programs. For example, you could group your Application Engine programs by product lines, such as Human Resources, Financials, Accounts Receivable, and so on.



Application Engine Project Tree in Project Workspace



For more information on managing projects, see *Using Application Designer*.

As with other definitions stored in a project, such as Records, the tree extends as far as the program name. You double-click the nodes within a project to open the associated definition.

Object Workspace

Application Designer presents your program using the following views:

- **Definition.** Use the Definition view to create and maintain your program.
- **Program Flow.** Use the Program Flow view to see the sequence of the runtime program execution. It appears in read-only mode.

This section introduces you to the purpose of each view and what you can expect to accomplish while it is active in Application Designer.

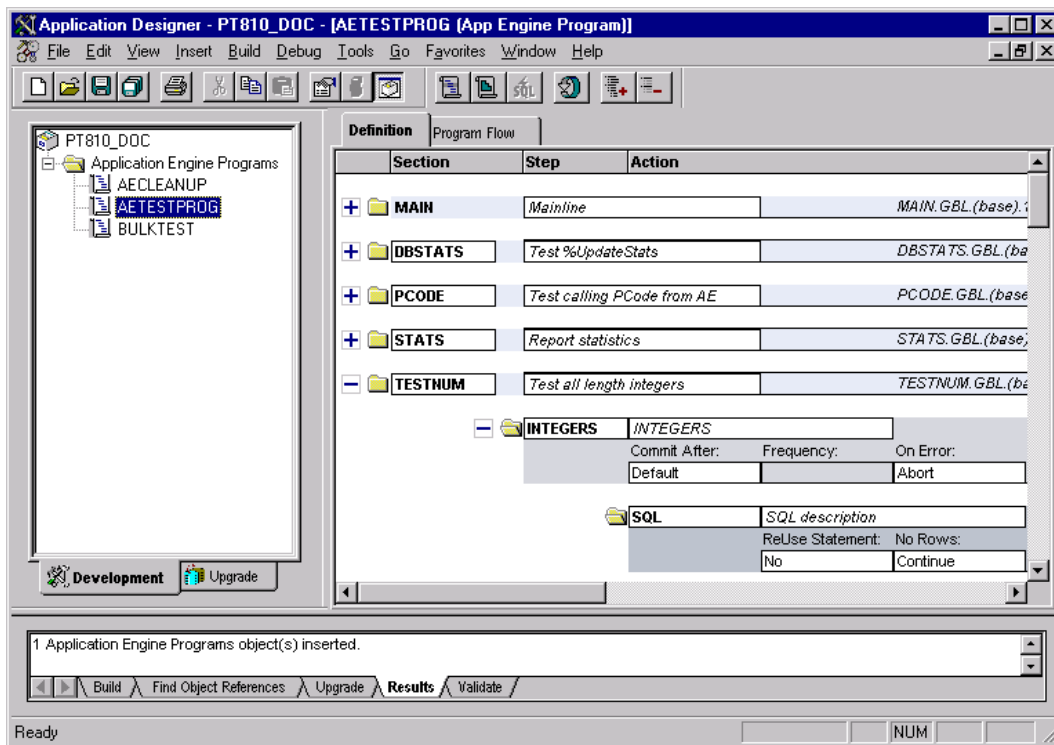
Definition View

The Definition view presents a collapsible “outline” or “tree” view of your Application Engine program. When you’re developing and defining your program, you will perform most of your work in the definition view.

The Definition view is where you create your *definition objects* within a defined hierarchical structure. A definition object is a Section, Step, or Action that you insert into your program and to which you apply unique properties. Within the metaphor of the Definition view, *nodes* represent the definition objects. A node is the visual representation of a Section, Step, or Action that you can select, collapse, modify, and so on.

The Sections that appear in the Definition view do not necessarily appear in the order that they will execute. To see the actual order in which the Sections, or the entire program for that matter,

executes, you need to switch to the Program Flow view. This is similar to COBOL where in the order that COBOL paragraphs are in does not indicate the order in which they are PERFORMed.



Application Designer's Definition View

Notice that when you first open an Application Engine program definition, all of the definition objects are collapsed. As you drill down to more detail you can view more of each definition object.

The following list presents some of the advantages the interface offers the development process:

- View all sections in a program in a single view.
- Collapse and expand definition objects (nodes) within the Definition view to hide and expose levels of detail regarding definition objects. To expand or collapse the subordinate nodes, click the "+" or "-" boxes that appear to the left of each definition object. Or you can expand or collapse all objects from the View menu.



For performance reasons Application Engine initially loads section objects with none of its child objects (steps and actions) expanded.

- Select a particular definition object in the view to highlight all rows that define the selected object. Once you've selected a node, all the appropriate menu items and toolbar buttons for that node are enabled or disabled.
- View details about a particular object by clicking the folder icon that appears to the right of the

object. You can also achieve the same results using the View menu.

- Add comments directly to objects.
- Display the related context, or popup menu for an item, just right-click on the object.
- Maintain definition attributes easily at each object level under the program definition using edit controls imbedded within the definition object. These controls include drop-down lists, edit boxes, and checkboxes.

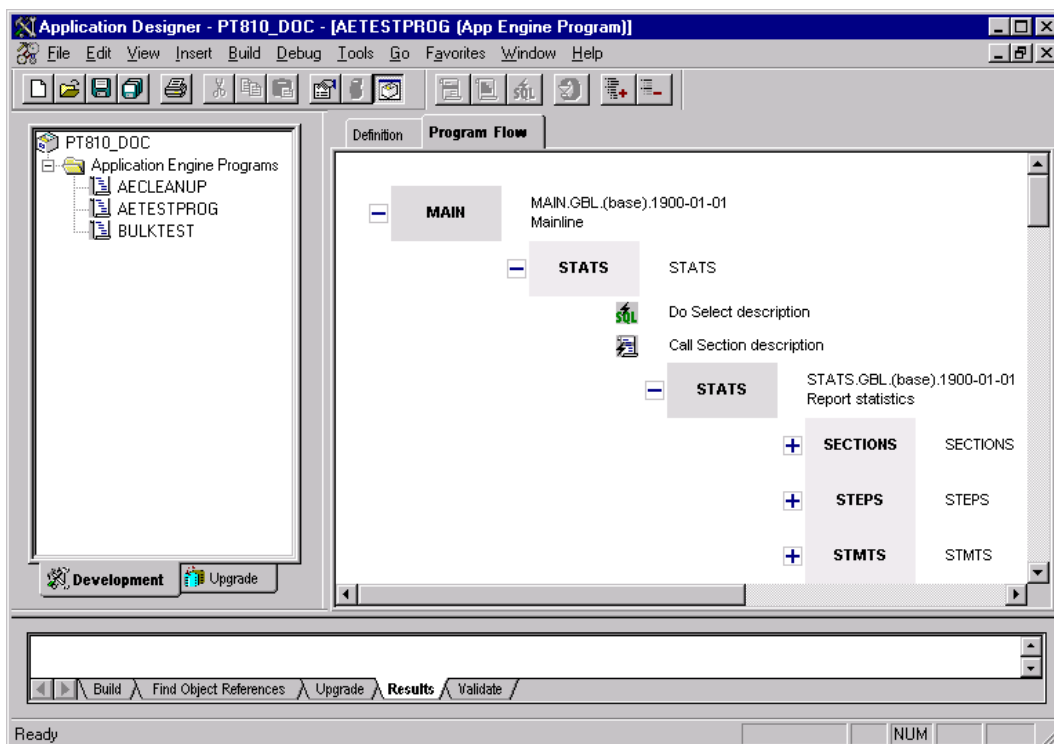


For more information on the interface see Menus.

Program Flow View

The Program Flow view reveals a program's logic flow as you've defined it in the Definition View.

It is a read-only view that enables you to see the expected sequence of steps to be executed at runtime for the program you are developing.



Application Designer's Program Flow View

As with the Definition view, you can control what's visible in the window by clicking the plus or minus sign boxes that appear next to each Section and Step. By using these controls you can control the amount of detail that appears for each definition object, as needed.

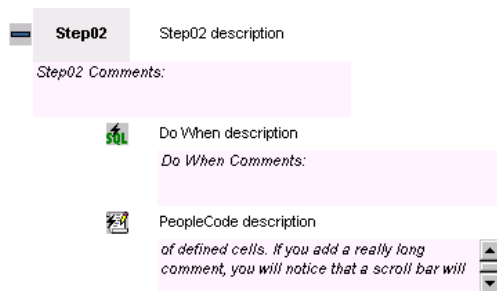
To display the pop-up menu for a node, right-click it. When using the popup menus, you do not have to select the node first.

You can also display the comments associated with definition objects by selecting **View, Show All Comments**, or for a particular node, right-click on it and select **Show Comment** from the pop-up menu.



For more information on the interface see Menus.

For comments or descriptions that are particularly long, a scroll bar will appear on the right-hand side of the edit box to help you read all the text.



Scroll Bar Appears for Long Comments

If you are in the Program Flow view, and you would like to view the SQL or the PeopleCode associated with a particular Action, you don't have to go back to the Definition View to see it. Just double-click the "row level" in which the Action appears, and the appropriate editor opens to display the code.

Navigation

As you see, there are only two views that you have to deal with while developing Application Engine programs, the Definition and Program Flow views, and, for the most part, getting from one to the other is relatively intuitive.

Despite that, there are a couple of features we've added to aid navigation that you'll want to keep in mind. One makes sure that you go straight to the node in the target view that you have currently selected in the definition view, and the other helps you easily locate all the occurrences of a particular object within the Program Flow view. You'll appreciate these features while working with large programs with numerous Sections and Steps.

Switching Between Definition and Program Flow Views

By default, navigation within either view will not affect the currently active row in the other view. This allows you to retain your place in one view while scrolling around in the other.

If you are switching from the Definition view to the Program Flow view, the first occurrence of the item (definition object) selected in the Definition view will appear in the Program flow view. The navigation logic considers that the Program Flow may reference the same item (definition object) more than once.

To switch between the two views, you can use any of the following methods described in the following sections.

View Tabs

As with any tabbed interface, if you click on the tab label, the associated view interface becomes active. When you return to the previous view, it will remain positioned on the current or last selected node within the program when you switched. This is true whether or not you've highlighted the item or just placed the cursor within an edit box.

View Menu

To switch between views using menu options, select **View, Jump to Program Flow** or **View, Jump to Definition** depending on the view that is currently active. When you select one of these menu options, the focus of the target view depends on what you have selected in the previous view. For example, if you have Section C, Step 4 highlighted in the Definition view, and you select **View, Jump to Program Flow**, Section C, Step 4 will be the focus of the Program Flow window. If the item selected resides in a program that is not already open, Application Engine will open the appropriate program, and the requested target will appear in the view window.

Pop-Up Menus

Using the pop-up menus associated with each view, you can achieve the equivalent result as the View menu option. To activate the pop-up menu, just right-click anywhere in either view. You'll, most likely use this method more frequently since the menu items are enabled and disabled according to the current node. So, it's quicker than using drop-down menus.

Within the Program Flow View

While you are in the Program Flow view, you can select the **Go to Next Reference** option from the popup menu.

When you select **Go to Next Reference**, the view switches to the next reference of a particular definition object if one exists. This helps you to quickly navigate through a program. For instance, if references to Section C, Step 4 appear three times because there are multiple calls to this object at runtime, you select **Go to Next Reference** to quickly and easily navigate to each reference, or call.

You can also select **Jump to this Definition**. This option enables you to go directly to the definition node in the Definition View that pertains to the current selection in the Program Flow view.

Menus

With any Windows application that you use, everything that an application is designed to do can be invoked by selecting a menu option. And, when ever you're using a new application it's always a good idea to become familiar with all the available menus and menu options.

This section is designed to present all of the applicable menus and menu items associated with Application Engine program development. After you've used Application Engine for a while, the location of the menu options should become second nature, but you may refer back here to double-check on the purpose or location a particular menu option.

With Application Engine you have full access to the typical Application Designer drop-down menus as well as a set of convenient context, or popup, menus for easy access to popular menu options. You will probably find that you use the popup menus more than the conventional menus for your menu selections.



Note. In the Definition View, all menu requests are relative to the highlighted, or selected, node. All context or popup menu requests are relative to the currently "clicked" node if there is no highlighted node.

Popup Menus

With each view, Definition or Program Flow, you have a different pop-up menu that you can use to quickly access popular menu selections. To invoke the pop-up menus, just right-click anywhere within either view.

The following topics describe the items on each pop-up menu.

Definition View Popup Menu

The following table contains each menu item on the popup menu that you'll see while the Definition view is active. Keep in mind that certain menu items are only enabled when a particular definition object is selected. For instance, only the menu items that apply to Sections appear when you have a Section selected.

<i>Item</i>	<i>Action</i>
View PeopleCode	Launches the PeopleCode Editor with the appropriate PeopleCode loaded. Enabled when a PeopleCode Action is selected.
View SQL	Launches the SQL Editor with the appropriate SQL loaded. Enabled when an Action containing SQL is selected.

Cut	Removes the selected item and copies it to a “clipboard”. Here, “clipboard” refers to a PeopleTools-only repository for sharing PeopleTools objects. You <i>can not</i> copy or paste into another program.
Copy	Copies the selected item.
Paste	Pastes the contents of the “clipboard” (the most recently cut or copied item) to the current location of the cursor.
Delete	Removes the currently selected node from the program definition.
Refresh View	Refreshes the current view and reorders the Definition objects as necessary.
Show Comment	Reveals the comments associated with the selected definition object.
Insert Section	Inserts a new Application Engine Section into the current program where the cursor is positioned. This option is only enabled when you have MAIN or another Section selected.
Insert Step/Action	Inserts a new Application Engine Step <i>and</i> Action within the currently selected Application Engine Section. This option is only enabled when you have a Section or a Step selected.
Insert Action	Inserts a new Application Engine Action within the currently selected Step. This option is only enabled when you have a Step or Action selected.
Jump to This Program Flow	Switches to the Program Flow view with the first occurrence of the currently selected definition object in focus. See Switching Between Definition and Program Flow Views.
Print	Prints the definitional form of the print output. It’s similar the definition view that appears, but it includes the details associated with the SQL and PeopleCode.

Program Flow Popup Menu

The following table contains each menu item on the popup menu that you’ll see while the Program Flow view is active.

Item	Action
Show Comment	Reveals the comments for a single definition object that appears in the Program Flow view.
Zoom In	Increases the size of the objects in the view.
Zoom Out	Decreases the size of the objects in the view.
Go to Next Reference	Moves focus to the next occurrence of a particular definition object in the Program Flow view. Some objects may appear multiple times within the program flow. This way you can quickly navigate to each occurrence. See Within the Program Flow View.
Jump to This Definition	Switches to the Definition view. The definition of currently selected node in the Program Flow view will be in focus. See Switching Between

	Definition and Program Flow Views.
Print	Prints the contents of the Program Flow view for review.

Dropdown Menus

The dropdown menus associated with Application Engine definition development are the File, Edit, View, and Insert menus. You'll find that the drop-down menus offer a richer selection of menu options that generally apply to all nodes in the current view.

In the following topics, we describe the menu options available to you as you use Application Engine's development environment. Many of the options are standard Application Designer menu options, but we'll tailor the description to Application Engine development where appropriate.

Keep in mind, that when you create the State and Temporary records (tables) associated with your Application Engine program, you'll be creating Field and Record objects. You design records in Application Designer using a different interface.

After you read this section, you will be familiar with all the menus and menu options available for your Application Engine development projects, and the more you use Application Engine the less you'll need to refer to this section.



Note. To reduce redundancy with content that already appears in the Application Designer documentation, we have excluded menu items that are common to all of Application Designer's interfaces. For the most part, we focus on menu options that are unique to the Application Engine development environment.

File Menu

The File menu offers the following options. All menu items are disabled or enabled according to the active view and the selected object.

Menu Item	Shortcut	Action
New	Ctrl+N	Displays the New dialog from which you can select <i>Application Engine Program</i> to begin developing a new program. When doing so, Application Designer automatically generates a default MAIN Section and initial "Step 01".
Open	Ctrl+O	Displays the Open Object dialog from which you can select <i>AE Program</i> to search for existing programs.
Object Properties		Opens the Application Engine Program Properties dialog where you can specify program-level properties. See Program Properties.
Rename (Object)		Displays the Rename Object dialog with Application Engine Programs selected as the Object Type .
Delete		Displays the Delete Object dialog from which you can delete

(Object)		previously saved Application Engine programs.
Page Setup		Displays the Page Setup dialog where you can select the options for your printed program definition.
Print Preview		Displays the layout in which your printed definition will appear. The window shows the currently active program. Use Page Setup to alter margins, content printed, borders, and so on.
Print	Ctrl+P	Displays the standard Print dialog where you can decide to send output to a designated printer or to a file. The content of the printed output will be that of the active definition.



Note. The menu items that do not appear in the previous list support actions that are consistent with the general Application Designer interface.



For more information on the Application Designer interface, see Using Application Designer.

Edit Menu

The Edit menu offers the following options. All menu items are disabled or enabled according to the active view and the selected object.

Menu Item	Shortcut	Action
Cut	Ctrl+X	Removes the selected item from the current definition and stores it for the next paste request. You can also cut and paste text.
Copy	Ctrl+C	Copies the selected node. You can also copy and paste text.
Paste	Ctrl+V	Pastes, or inserts, a previously copied object into the definition at the current position of the cursor.
Rename (node)		Highlights the name of the selected node allowing you to easily edit the node label.
Delete (node)	Del	Deletes the selected node.
Validate All Sections		This option provides a toggle switch for the automatic Section validation that occurs upon the Save event. When Validate All Sections is selected, Application Engine validates each Section. In some cases, you may not want to validate each Section with each Save, as it is unnecessary and can hamper performance with larger programs. When Validate All Sections is <i>not</i> selected, Application Engine only validates the Sections that you have modified since the

		previous Save.
Validate Program Now		This option provides an "on demand" validation for the entire program as opposed to one that you invoke through the Save event. Typically, you'll use this when you do not have Validate All Sections selected.
Find Object References		This option allows you to find all of the references to a particular section within a single database. In this case, "references" mean Call Section actions that call a selected program.
Go To Section	Ctrl+G	Use this option for easy navigation to a selected section within a program. This is particularly useful within large, more complex programs.



Note. The menu items that do not appear in the previous list support actions that are consistent with the general Application Designer interface.



For more information on the Application Designer interface, see Using Application Designer.

View Menu

The View menu includes additional menu items to support the Application Engine interface. All menu items are disabled or enabled according to the active view and the selected object.

Menu Item	Shortcut	Associated Action:
PeopleCode		Launches the PeopleCode Editor with the appropriate PeopleCode loaded.
SQL		Launches the SQL Editor with the appropriate SQL loaded.
Show All Comments		Displays all the comments associated with the nodes.
Hide All Comments		Hides all the comments associated with the nodes.
Expand All	Ctrl+Shift+X	Expands all collapsed Sections to display all the Steps in current view and expands all Steps to display all Actions.
Collapse All	Ctrl+Shift+C	Collapses all expanded Sections and Steps in program. All that will appear are Sections.
Show All Details		Expands all collapsed node details in current

Menu Item	Shortcut	Associated Action:
		view, so that no rows defining node-level properties are hidden.
Hide All Details		Collapses all node detail rows.
Refresh	F5	Refreshes the current view, logically reordering all definition objects as needed. For more information See Refreshing the Workspace.
Section Filtering		No Filtering: Displays Definition Filter dialog to define filtering options for the current view. Default: Display Filter dialog to define default filtering options for the current operator. Custom: Display Filter dialog to define custom filtering options for the current view (see section View Filter). For more information see Section Filtering
Zoom In	F8	Enlarges the contents of the current view to display less detail in the Object Workspace.
Zoom Out	F9	Shrinks the contents of the current view to display more details in the Object Workspace.
Jump to ... Program Flow/Definition		Toggles the current view. If you're in Definition view, you'll switch to the selected node in the Program Flow view and if you're in Program Flow view, you'll switch to the selected node in the Definition view. See Navigation.



Note. The menu items that do not appear in the previous list support actions that are consistent with the general Application Designer interface.



For more information on the Application Designer interface, see Using Application Designer.

Refreshing the Workspace

As you are working on an Application Engine program you will be inserting, renaming, and deleting objects from the program. In a large program, it can be easy to lose your place or become

disoriented. The Refresh option will reorder all the nodes for the current definition according to the following logic:

- The MAIN Section, if defined, is always displayed first. The remaining Sections appear alphabetically by name, which makes it easier to locate a given Section within the Program definition. Keep in mind that the system, at runtime, executes Sections through Call Section Actions within Steps, not by the order in which they are defined.
- Steps are never automatically reordered in the Definition view, and, at run time, they execute in the sequence in which you define them.
- Actions are always logically reordered based on their Action type, which defines their runtime sequence.



Note. When you save a modified definition, the system automatically refreshes the view.



For more information on the types of actions and the execution hierarchy, see [Actions](#).

Insert Menu

You use the Insert menu to add Sections, Steps, and Actions to the current definition, and you can add both Program and Section-level objects to the current project using the Insert menu.

The following table presents the menu options that appear on the Insert menu for Application Engine.

<i>Menu Item</i>	<i>Shortcut</i>	<i>Action</i>
Section	Ctrl+Shift+S	Inserts a new Section node into the current program after the selected node. Application Designer inserts sections beneath the currently selected node. If you do not have a current node selected, the system inserts the new section as the last object in the definition.
Step/Action	Ctrl+Shift+T	Inserts a new Step and a default SQL Action node after the currently selected Section or Step. Enabled in Definition View when a Step or a Section is selected.
Action	Ctrl+Shift+A	Inserts a new Action type after the currently selected Step or Action. Enabled in Definition View when an Action or a Step is selected.
Current Object into Project	F7	Inserts into the Project the current Program and optional Sections from the current program definition.

Menu Item	Shortcut	Action
Objects into Project	Ctrl+F7	Prompts you specify Application Engine objects to insert into the current Project.

When you've elected to automatically insert modified objects into current project at save time (Tools, Options, Project), Application Engine does not insert the modified Sections into the project if the project *already* contains a delete request for an object with the same key values. Application Engine inserts any delete requests for a given Section into the current project, regardless of the Tools, Options setting in Application Designer.










Let's say you delete a Section node from the current Application Engine program, and then you re-insert a Section node and rename it to the same name as the Section you just deleted. In this case, the Section object *will not be* inserted into the project regardless of your Tools, Options setting. This is due to the fact that a delete action already exists for this object. To resolve this situation you'll need to either manually remove the delete request before inserting the new Copy request or manually reset the proper flags in the Upgrade project that changes the action type from Delete to Copy.








Toolbar Buttons

As stated previously, the Application Designer interface, for the most part, remains the same regardless of what designer is currently active. So, if you've developed other applications using PeopleTools, you can use the same Application Designer toolbar buttons whether you're designing a page or a batch program and expect similar results.

Keep in mind that when Application Engine Designer is the active designer, some of the normal toolbar buttons that you use in Application Designer, such as the button associated with SQL Build, will be disabled since it is not relevant to your Application Engine program.

The following table presents the toolbar buttons you'll use as part of the Application Engine Designer interface.

Button	Action
	Creates a new Application Engine program.
	Opens an existing Application Engine program.
	Saves the active Application Engine program.
	Saves all open Application Engine programs.
	Prints the definition of the active Application Engine program for review.
	Cuts currently selected node and all it's subordinate objects.
	Copies currently selected node and all it's subordinate objects.
	Pastes currently selected node and all it's subordinate objects after the selected node.
	Shows the active object's properties. This opens the Application Engine

Button	Action
	Program Properties dialog with the General, State Records, and Advanced tabs.
	Toggles the display for the Project Workspace window. This can help you to gain a better or more complete view of your Application Engine program definition. You'll just see the Definition View or Program Flow view in the Object Workspace without the tree view of the Project Workspace.
	Inserts a Section after the currently selected section or at the end of the program definition if you don't have a section selected.
	Inserts a Step and an Action.
	Inserts an Action after the currently selected Step.
	If necessary, refreshes the current view and reorders the nodes as needed.
	Expands all collapsed nodes in current view.
	Collapses all expanded nodes in current view.

Designer Interface Tips



The following topics describe some basic procedures with which you will want to become familiar before you begin developing your Application Engine program. The Application Designer interface is very intuitive, and many of these procedures you can figure out on your own, but reading about them here may save you some time and questions down the road.

Viewing Definition Objects

You won't always need to see every property of every definition object currently in view. Sometimes you may just need to see the names and descriptions of multiple definition objects to get a high-level view of your program or of a particular Section or Step. Here are two tips to help you take advantage of hiding and showing different elements of your definition objects.

Using the Plus and Minus Signs

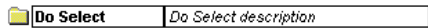
You have the ability to control whether subordinate objects appear beneath a Section or a Step in the Definition view.

- To expand a Section or a Step showing the objects it contains click .
- To collapse a Section or a Step, click .

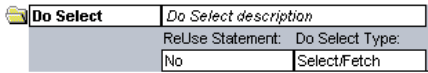
Using the Folder Icons

The folder icons apply to all objects in the definition view. You'll see them just to the left of the object name. To open or close the folder, just click it.

If you only want to see the object name and its description, close the folder.



Or, if you want to display the detail properties associated with a particular object, click its folder to open.



Section Filtering

Both the Definition and the Program Flow views support the ability to filter the view contents by allowing the user to determine whether or not the view will contain all definition objects, or only those objects in which the developer is interested.

The Section Filtering menu options allow you to filter the current view so that you only see Sections and Steps based on specified criteria. This feature will typically be used in situations where you are developing Application Engine programs intended to run in multiple markets, on multiple platforms, or with effective dated Sections. If you are developing small programs designed to run against all database types or in all markets using the same code, you will probably not need to use this feature as much.

To enable or modify the filtering options, select **View, Section Filtering**. The Definition Filter dialog offers the following filtering options:

- **No Filtering.** Select this option if you want to see all objects in your program regardless of any Section attributes, such as Market, Effective Date, and so on.
- **Default.** Displays the Definition Filter so that you can view the values that comprise the Default filtering criteria. If you change them and click OK, you have defined a Custom filter.
- **Custom.** Displays the Definition Filter dialog so that you can define custom filtering options for the current view.

If you select **Section Filtering, Default** or **Section Filtering, Custom**, the following dialog will appear:

Definition Filter

Section

Market: ITA

Platform: Microsoft

As of Date: 2000-04-11

April 2000

Su	Mo	Tu	We	Th	Fr	Sa
26	27	28	29	30	31	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	1	2	3	4	5	6

☒ Active Status

OK Cancel

Definition Filter

In this example, only definitions that represent the following criteria will appear in the Definition and Program Flow views.

- Sections pertaining to the Italian market (Market code “ITA”).
- Sections that are defined for the Microsoft SQL Server platform..
- Sections with and As of Date equal to or greater than April 7th, 2000.
- Sections that have their Active flag enabled.

While the settings on the Definition Filter dialog are active, all the definition objects that do not meet the filtering criteria will be screened, or filtered, from view. If you wanted to see all Market-related definitions for a Program, you could update the default profile, or define a custom Filter, selecting (*none*) from the **Market** drop-down list.

There is a separate Filter attribute for each of the Definition and the Program Flow views. For example, you could update the Definition view filter to only show objects defined to run on an Oracle platform. Switching to the Program Flow view using the view window tabs would not cause the view filter in the Program Flow view to be reset. However, if you use the *Jump to...* menu option, the target view’s Filter will match the original Filter’s attributes offering consistency between the two views.

Selecting the menu item, No Filtering, causes the current Definition view to essentially ignore the current Filter attributes and display *all* the objects defined for the current program. This allows the user to see the entire definition—all Markets, all platforms, active and inactive, regardless of As of Date. When Default or Custom Filtering is selected, the prior filter attributes for the active Program are restored.

The following table represents all Section Filter-related values and their attributes.



Note. All filtering options only pertain to Section-level nodes.

Filter	Values	Description
Market	GBL (default) <Market>	GBL. Displays only Sections defined with the default Market code of GBL (global). <Market>. View displays only Sections defined by the selected <Market> code.
Platform	DB2/MVS DB2/Unix Informix Oracle Microsoft Sybase	(default). Displays only those Sections defined to be database platform independent (default platform) <Platform>. View only displays Sections defined for the selected database platform, as in Oracle or DB2/MVS.
As of Date	1900-01-01 (default) <Date>	(none). View displays all sections regardless of their effective date. <Date>. View only displays Sections that are effective as of the specified <date>.
Active Status	Active (default) Inactive	Active. View only displays Sections defined as Active. Inactive. View displays all Sections regardless of their Active status.

Activating Object Property Edit Boxes

The Definition view is designed so that you can specify all of the necessary properties for all of your definition objects, such as Sections and Steps. As you've seen in previous examples, all of the properties for definition objects can be specified on the actual node of the definition object. To alter or add any of the values for a particular definition object you first need to "activate" the control by positioning the cursor, or clicking, within the appropriate edit box.

The following topics briefly describe the controls you'll encounter as you modify definition properties.

Text Boxes

For Text boxes such as the Section and Step name, object descriptions, and all the Comments boxes, just click in the appropriate text box to activate the control, position the cursor in the correct location, and begin typing. You can also select text to delete or replace.

You can always tell which edit box is active by the dotted line around the circumference of the box.



Drop-Down Lists

For the most part, you'll be selecting values from drop-down list boxes.

When you click inside or tab into an edit box that contains a drop-down list, as shown:



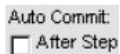
a down arrow will appear on the right-hand side of the control.



You could also click on the right side of the edit box, where the hidden arrow is located, to display the dropdown list with just a single-click.

Toggles/Switches

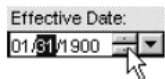
Some of the properties are a simple toggle or switch. For instance, a Section can either commit after a given Step or not at all. A check box toggles the Commit property off and on. For example, currently this Section's Auto Commit control is not selected, or checked, and therefore commit is disabled.



To enable the Section-level commit, just select the checkbox. If a checkbox is active, you can use the Spacebar to select and deselect the checkbox.

Calendars

When setting the Effective Dates for Sections, you'll see the standard calendar control that appears throughout our applications. If you click in the edit box, two controls appear on the right. Use the inside control to "spin" the selected date value for small increments.



Use the outside control, or down arrow, to invoke the graphical calendar control, and click to select a date.



Selecting Objects

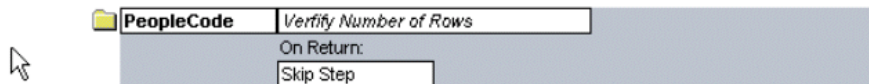
When you have a definition object selected, such as a Section or Action, it becomes highlighted and the node colors are reversed. Once you've selected a definition object, all of the menu options relevant to the selected object will become enabled. A selected, or highlighted, definition object appears as follows:



Note. In subsequent chapters, when instructions say to “select” a definition object, the prior example illustrates a “selected” object.

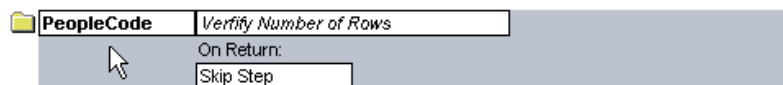
To select a definition object, do one of the following:

- Click in the left margin of the Object Workspace directly across from the definition object.



Clicking in the Margin

- Click on the definition object itself.

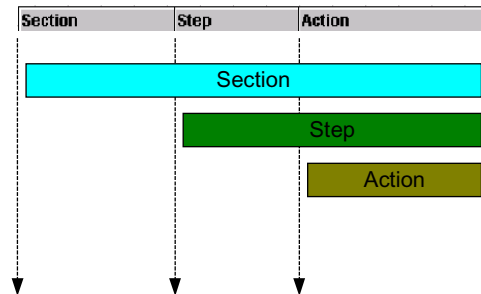


Clicking on the Definition Object

Definition Object Layout

As you'll see after you work with the Application Engine Designer, the definition objects, as in Sections, Steps, and Actions, are positioned along three invisible columns or vertical lines. Notice that in the Definition view, just under the tabs, there is a black strip with white text that indicates each of the different columns.

The definition objects are arranged according to these columns. For instance, only Sections extend as far as the Section column, and Actions only extend as far as the Action column. The following example illustrates how the definition objects are arranged.



Definition Objects Positioned along Columns

This helps you to visually keep track of the numerous definition objects that you will add to your program.

Working with Definition Objects

The following procedures describe the basic steps that you'll use while working with definition objects. These procedures apply to all of the definition objects, be they Section, Step, or Action. Procedures that apply to a specific type of definition object type appear where that definition object is discussed.

To copy a definition object

1. Select the definition object, or right click to open the popup menu.
2. Perform one of the following:
 - Select **Copy** from the popup menu.
 - Select **Edit, Copy** from the dropdown menu.
3. Position the cursor where you would like the copied definition object to reside and perform one of the following:
 - Right-click, and select **Paste** from the popup menu. You can only paste an object to its appropriate place within the object hierarchy. For example, you can't paste an Action immediately after a section because Actions are subordinate to a Step.
 - Select **Edit, Paste** from the dropdown menu.



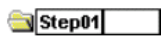
Note. The Application Engine Designer does not currently support copying definition objects between separate programs.

To move a definition object

1. Select the definition object, or right click the object to open the popup menu.
2. Perform one of the following:
 - Select **Cut** from the popup menu.
 - Select **Edit, Cut** from the dropdown menu.
3. Position the cursor in the target location within the program where you'd like the definition object to reside after performing one of the following:
 - Right-click and select **Paste** from the popup menu.
 - Select **Edit, Paste** from the dropdown menu.

To rename a Section or Step

1. Click in the Name edit box of the definition object you want to rename.



2. Double-click to select the entire name for removal, or backspace over any individual characters you want to remove.



3. Enter the new name.



To change an Action type

1. Click on or tab into the Action edit box to activate it.
2. Perform one of the following:
 - Select the appropriate Action from the dropdown list.
 - Enter one or two unique characters for a valid Action type and tab out of the edit box. For example, if you just enter *Pe* and tab out of the edit box, the PeopleCode Action type automatically appears.

To delete a definition object

1. Select the definition object, or right-click it.
2. Perform one of the following:
 - Right-click and select **Delete** from the popup menu.
 - Select **Edit, Delete** from the dropdown menu.

3. Respond appropriately to the Application Designer prompts to confirm the deletion of a definition object.

To clone a definition object

1. Copy the definition object as described in To copy a definition object.
2. Move the definition object as described in To move a definition object.
3. Rename the definition object as described in To rename a definition object.

CHAPTER 3

Creating Application Engine Definitions

In the previous chapters we have become familiar with why you use Application Engine, the development environment in which you define your Application Engine programs, and the different definition objects and their properties. In this chapter, we'll cover the basic procedures involved with inserting and maintaining definition objects. Then in the following chapter, we'll discuss the overall process of designing, building, and testing your program, as well as some of the more subtle aspects of effective Application Engine programming.

This chapter will cover all the procedures related to inserting and specifying properties for all the definition objects from entire programs to Actions. The topics discussed in this chapter will be useful in the subsequent chapters, as you delve into the finer details of building batch programs with Application Engine.

After reading this chapter you will be familiar with the following:

- Creating new programs and specifying program-level properties.
- Inserting Sections, Steps, and Actions and selecting the appropriate properties for each.
- Performing basic procedures that apply to all definition objects, such as cloning, deleting, moving, and so on.
- The hierarchy in which Actions are arranged and executed.



Note. The following sections assume that you have PeopleTools running and the Application Designer open. Also, we begin our discussion at the Program level and work toward Actions. Although, we encourage you to store your Application Engine program definitions within Application Designer Projects, we do not document managing Projects here.



For more information on Application Designer Projects, see Using Application Designer Projects.

Programs

An Application Engine program is comprised of the logically ordered set of Sections, Steps, and Actions defined within it. An executable program must contain at least one Section, called MAIN, used to identify the starting point of the program. Keep in mind that a Section should

contain at least one Step and each Step should contain at least one Action. Therefore, the minimum that a program requires to execute is at least one of each definition object. Typically, your programs contain numerous Sections, each containing one or more Steps and Actions.



Note. Application Engine program definitions can exist without a MAIN Section, but only when you define them as an Application Library. Libraries contain shared Sections that other programs call. They can't be executed directly because they have no MAIN Section.



For more information on Application Libraries, see Temp Tables.

Just as you maintain the properties and attributes of individual Sections and Steps, you can also maintain more general, far reaching attributes that affect the program as a whole, such as specifying the State Records that a program uses. This section covers the properties that you can specify at the program level. Keep in mind, for each definition object that you add, such as Sections and Steps, you will have additional properties to specify at the Section level and Step level.

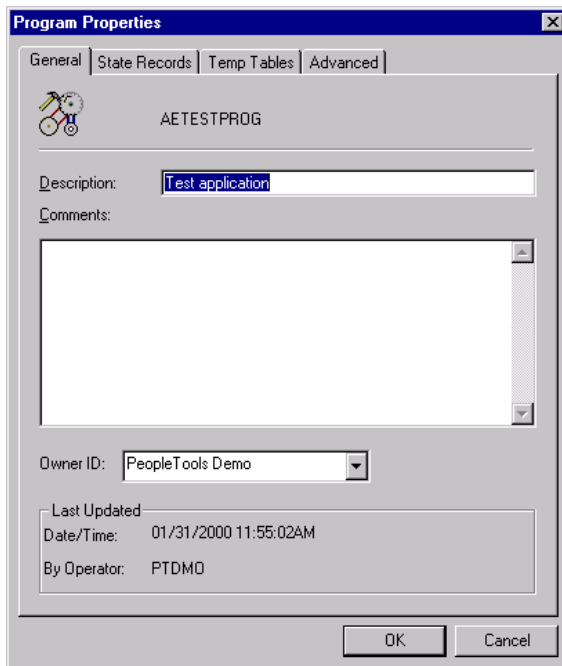


For more information on opening programs or creating a new program, see Procedures Related to Programs.

Program Properties

When you have an Application Engine program open in the Application Designer you can view and modify the properties assigned to an entire program just as you would a Step or a Section. If you want to view or modify the properties associated with a program, press the Properties button or select File, Object Properties while the program is open. You can also press ALT-ENTER.

After doing so, the Program Properties dialog appears.



Program Properties Dialog Box

The following sections describe the controls and options that each tab offers on the Program Properties dialog. This is where you can view and modify program-level properties.

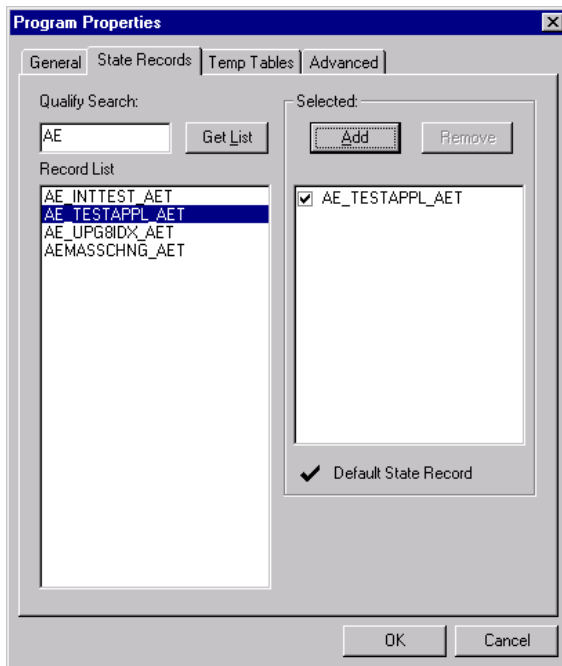
General

On the General tab you can specify certain identification values for your Application Engine program. You can specify the following:

- **Description.** Enter a descriptive name for your program. This edit box has a 30-character limit.
- **Comments.** Here you can describe what the program is designed to do or add any other useful comments that development or administrative staff may need to see.
- **Owner ID.** Enter the Owner ID for the program. Owner ID is a way to identify which objects are owned by which PeopleSoft products, such as General Ledger, Accounts Receivables, and so on. The values in the dropdown list are translate table values associated with the OBJECTOWNERID field. This field is entirely optional.

State Records

The **State Records** tab is where you specify and maintain the list of State records designed for a particular Application Engine program.



State Records Tab

The State Records tab contains the following controls:

- **Qualify Search.** This edit box allows you to enter any wild card characters or complete table names to limit the results that will appear in the Record List. By default, the Record List box will contain all record names that end with the extension AET. This extension identifies the record within the system as an Application Engine record.
- **Get List.** Press this button, to populate the Record List box. You'll want to enter any qualification text in the Qualify Search edit box prior to pressing this button.
- **Record List.** This text box contains the results of your State Record search.



For more information on developing and designing State Records see, State Records.

- **Selected.** This group box is where you select State Records for use with a particular program. Use the Add button to include selected records that appear in the Record List. Use the Remove button to exclude selected records that appear in the Selected list box. Indicate which State Record will act as the default State Record by selecting its checkbox. For your default State Record, you only need to reference fieldnames in your PeopleCode and SQL (for the active program). When you reference a non-default State Record, you do so by using *recname.fieldname*.

Temp Tables

Temporary Tables, also referred to as Temp Tables, can be a very important asset for many of your Application Engine programs. Typically, temporary tables store transient or "intermediate results" during a program run.

You also use Temporary Tables to improve performance. For example, if you find that multiple times during a run the program accesses a small subset of rows from a much larger table, you can insert the necessary rows into a temporary table as an initialization task. Then the program accesses the data residing in the smaller temporary table rather than the large application table. This technique is similar to reading the data into an array in memory, except that the data never leaves the database, which is an important consideration when the program employs a set-based processing algorithm.

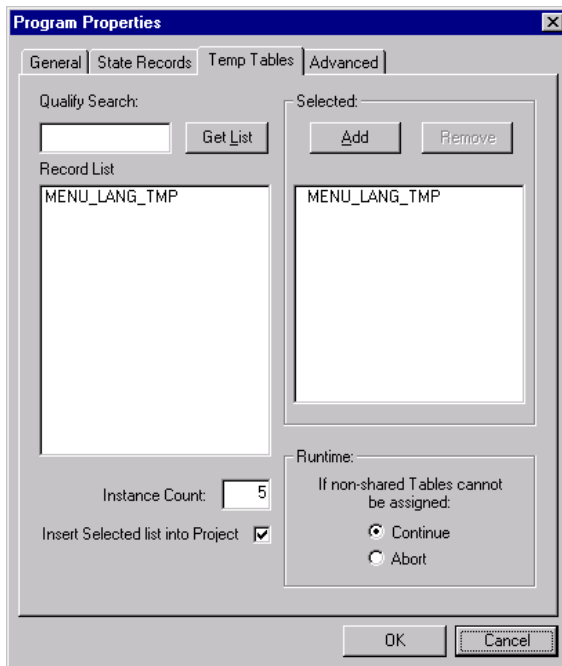


For more information on set processing, see Set Processing.

Any number of programs, not just your Application Engine programs, can use the temporary table definitions. What PeopleTools offers with the Temp Tables tab is the ability to dedicate physical table instances to one, and only one, Application Engine program at runtime. When you specify a temporary table on the Temp Tables tab, Application Engine automatically manages the assignment of temporary table instances. When Application Engine "manages" a dedicated temporary table instance, it controls the locking of the table before use and the unlocking of the table after use.



Note. You must have already defined the required temporary tables prior to associating them with an Application Engine program.



Temp Tables

The Temp Tables tab contains the following controls:

- **Qualify Search.** This edit box allows you to enter any wild card characters or complete table names to limit the results that will appear in the Record List. By default, the Record List box only will contain records that are of type "Temporary Table". This is an attribute applied at the time you create the Record in Application Designer.
- **Get List.** Press this button, to populate the Record List box. You'll want to enter any qualification text in the Qualify Search edit box to limit the search prior to pressing this button.
- **Record List.** This text box contains the results of your Temp Table search.
- **Selected.** This group box is where you select temporary tables for use with a particular program. Use the Add button to include selected records that appear in the Record list. Use the Remove button to exclude selected records that appear in the Selected list box.
- **Instance Count.** The Instance Count value controls the number of physical tables to be created for each dedicated table for this program during the SQL Build procedure in Application Designer. Typically, you would set this number to equal the number of parallel program runs that you anticipate. For instance if you expect up to five instances of the same program to run simultaneously, then you would set the Instance Count to 5.
- **Insert Selected List into Project.** If the active Application Engine program definition belongs to a Project, then you can opt to include the dedicated temporary tables for this program within the same project.
- **Runtime.** When you are implementing dedicated temporary tables, you must consider how your program should behave in the event the number of active processes exceeds the number of dedicated temporary table instances. Do you want the program to abort, or do you want it to use the base table (PS_recname with no suffix), which is a non-dedicated table. The Runtime

options enable you to control how an Application Engine program behaves in the event that an instance of its specified dedicated temporary tables are not available. If you select **Continue**, then Application Engine uses the base version, or undedicated version, of the temporary tables. If you select **Abort**, then the program exits with a meaningful error message.



As long as the table is keyed by PROCESS_INSTANCE, and the application SQL includes the Process Instance in the WHERE clause, then the table can be shared by multiple processes. The best performance, however, occurs when a program runs against a dedicated temporary table instance.

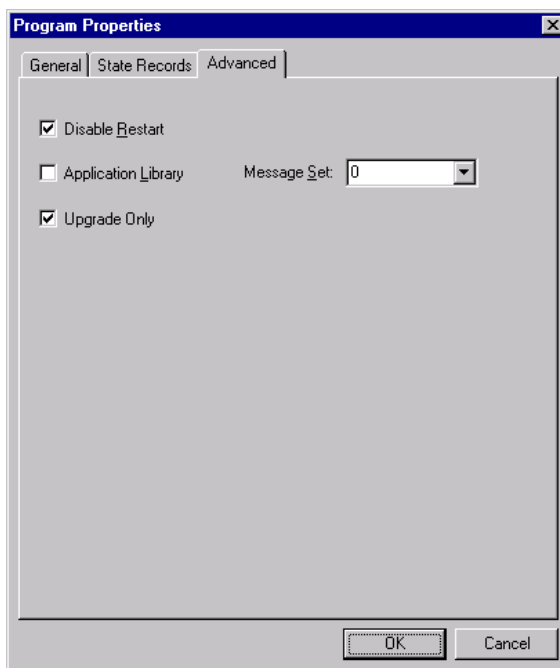
Using temporary tables for parallel processing--one of the more subtle aspects of Application Engine programming--is discussed in more detail later. Temporary tables are also extensively used in a development approach referred to as set processing.



For more information on developing and using temporary tables for use with your Application Engine program see Using Temporary Tables and for set processing information see Set Processing.

Advanced

The Advanced tab offers the remaining program-level options.



Advanced Tab

The Advanced tab allows you to control the following:

- **Disable Restart.** As stated, Application Engine has built-in restart capabilities. In many cases, you will want to have Restart enabled. However, there are some situations, which are discussed later, where you may want to disable Restart. To disable restart for a particular program, select **Disable Restart**.



For more information on Restart, see Developing for Restart.

- **Application Library.** Most Application Engine programs are defined to be executable. An executable program has a main entry point in its definition: the Section MAIN. This defines the entry point so whatever method you choose to invoke the program, such as command line or Process Scheduler, can initiate the program. However, in some cases, you may want a program to only contain a collection, or “library,” of common routines (in the form of “callable” Sections) that you do not want to run as a standalone program. When sections are defined as “Public,” other programs can call the Sections, or routines, that exist in the “library” at runtime. Because this type of Program is not designed to run as a stand-alone program, it does not require the MAIN Section, or initial entry point. Setting the **Application Library** option, renames or removes any existing MAIN Section.



Note. Application Libraries are the appropriate spot to store a collection of shared Application Engine program Sections. It is not intended to store a specific SQL Action with in a Section. To share common SQL, use the SQL repository.

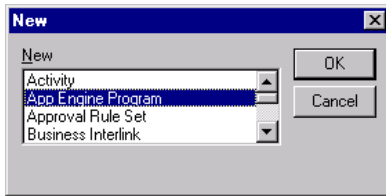
- **Upgrade Only.** This option is for programs intended to be used for upgrade purposes only. PeopleSoft delivers some Application Engine programs for upgrade-only uses. Unless you are responsible for the upgrade procedure at your site, you won’t need to deal with these programs. These programs are documented specifically within the Upgrade documentation accompanying your shipment.
- **Message Set.** Specify the message set value that you want assigned as the default message set number for this Application Engine program. The system uses this Message Set value for all Log Message Actions where the message set isn’t specified.

Procedures Related to Programs

The following procedures apply to the program level of your Application Engine development.

To create a new program definition

1. Select **File, New**, or press CTRL + N.
2. From the **New** dialog, select *App Engine Program*, and press **OK**.

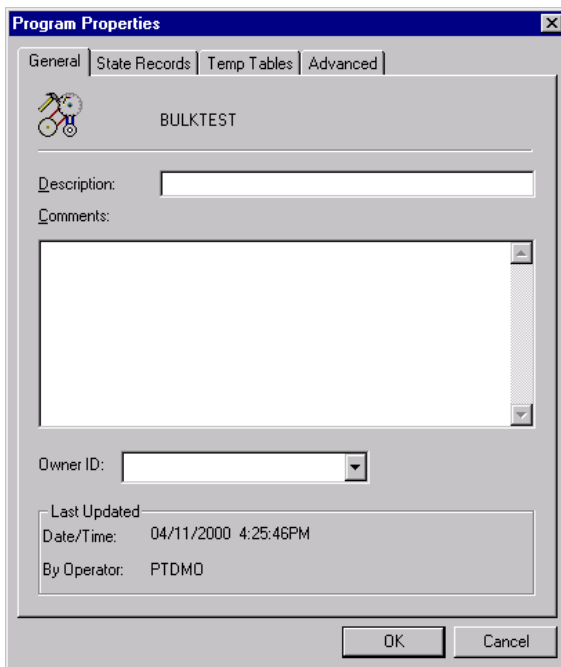


New Dialog



Note. Whenever you create a new Application Engine program definition, the designer will always create a default MAIN Section node and a Step node (Step 01). This provides you with a starting point for building your new Application Engine program.

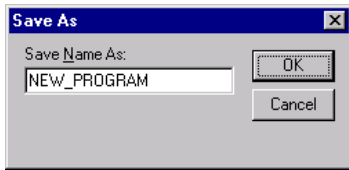
3. Specify the appropriate values in the Program Properties dialog.



Program Properties Dialog

After selecting the appropriate properties, press **OK**.

4. Save and name your program.
5. Enter the name of your program in the **Save Name As** edit box, and press **OK** to return to the Definition view.



Save As Dialog

To open an existing program


1. Select **File, Open**.
2. On the Open Object dialog, select *App Engine Program* from the Object Type drop-down list.

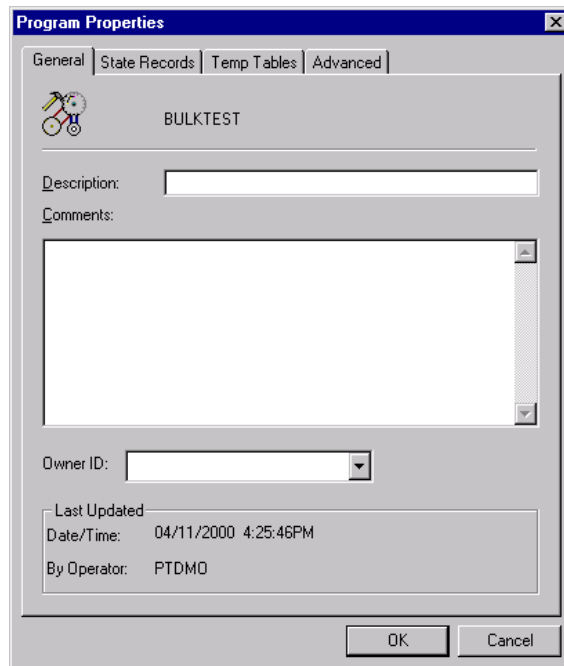
You can also select Project from the Object Type drop-down list if your program resides in a Project.



For more information about opening objects in Application Designer, see Using Application Designer.

To view or modify program properties

1. With the program active on the desktop, perform one of the following:
 - Select File, Object Properties.
 - Press .
 - Press ALT-ENTER.
2. On the Program Properties dialog, select the appropriate tab for viewing or modifying; you have the following choices:
 - General
 - State Records
 - Temp Tables
 - Advanced



Program Properties, General Tab

To copy a program

1. Open the program as described in To open an existing program.
2. Specify any new, desired properties for the new program in the Program Properties dialog, and press OK.
3. Select File, Save As.
4. Enter the new program name in the Save Name As edit box on the Save As dialog, and press OK.

To rename a program

1. Select File, Rename.
2. On the Rename Object dialog, make sure *App Engine Program* appears as the Object Type.
3. In the box that contains the results of your search, click on the program you want to rename.

Name	Description
APPM5GARCH	Application Messaging Archive
PT800GES	
PT800_B7	

4. Click Rename.

5. Place the cursor in the box that appears around the highlighted program name.

PT800_B7

6. Enter the new name for the program.

PT800_B8

7. Click Rename again, and respond appropriately on the Confirm Rename dialog.

All static references in other programs to the renamed program, the system modifies automatically. For instance, if you call the renamed program from another Application Engine program, the Call Section Action in the calling program will be modified to reflect the new program name. All the Sections and Steps will be saved under the new name. Only one occurrence of any name can exist.



Note. If the renamed program gets called in a Dynamic Do, the reference will not be automatically modified. You should also, manually check and modify any references to the new program name in CallAppEngine or other PeopleCode functions.

To delete a program

1. Select File, Delete.
2. In the Delete Object dialog, select *App Engine Program* from the Object Type drop-down list.
3. Specify any applicable Selection Criteria, and click Delete.
4. From the search results, select the program you want to delete, and click Delete.
5. Respond appropriately on the Confirm Delete dialog.

Sections

An Application Engine program Section is comprised of one or more Steps and is somewhat equivalent to a COBOL paragraph. If you are familiar with PeopleCode, you can also think of a Section as equivalent to a PeopleCode function. You can create Sections that are platform-independent or platform-specific, intended for a particular Market, and Effective Dated.

Whenever you create a new program, you simultaneously create a Section, called MAIN. PeopleSoft automatically include the MAIN Section in each program since every (executable/standalone) Application Engine program requires at least one Section, the MAIN Section. The MAIN Section identifies the program's entry point so that it can be invoked or called by another program. A “program” that is intended to serve as an Application Library, does not contain a MAIN Section because its Sections are only executed when called from external programs or sections within the same library.

All new programs already have one Section present to get you started, and, in most cases, you'll want to add additional Sections to a program. If you have are creating a large batch program, it is likely to require multiple Sections containing a wide variety of Steps and Actions. PeopleSoft recommends that you keep your programs as simple as possible. You can keep it simple by developing modular programs and using common library routines.


After reading this section, you will be familiar with the following:

- Inserting Sections into your program.
- Defining Section properties.
- Copying, renaming, and deleting Sections.

For some of the options that you will encounter while adding Sections you will be referred to other topics in this PeopleBook. For instance, the *full* discussion of Application Engine's Commit does not occur within the procedure that describes inserting a Section. This helps to keep information that is relevant to many aspects of Application Engine in one main location where it can be fully discussed. Also, having the bulk of the detailed information in one place keeps more complicated and subtle aspects of Application Engine in a separate location that you can go to after you've become familiar with the fundamentals.

Section Properties

You can find all of the controls that specify Section properties in the Definition view. For example, for each Section included in your program there will be a node, as shown in the following example, from which you specify all of the attributes that you'd like to associate with a particular Section.

 MAIN	MAIN description		MAIN GBL (base), 1900-01-01				
Market:	Platform:	Effective Date:	Effective Status:	Section Type:	Auto Commit:	Access:	
GBL	(base)	01/01/1900	Active	Prepare Only	<input type="checkbox"/> After Step	<input type="checkbox"/> Public	

Section Object



Note. The previous example happens to depict the MAIN Section, but the properties that you specify for the MAIN Section are equivalent to the properties you specify for any subsequent Sections.

The values you specify at the Section level, for the most part, apply to all the objects contained within that Section. You can, however, override the Auto Commit level at the Step level. With the Effective Status, you can determine whether the Step itself is active or not. You can inactivate a Step in an active Section, but you can't activate a Step in an inactive Section. The system considers the "active" step, because the owning Section is set to inactive.



For more information on Step properties, see [Step Properties](#) , and for more information on Commit, see [Application Engine Commit](#).

The following table contains a brief description of each option.

Control	Values	Details
(Description)	<Section Name> description	In this edit box you can enter a more descriptive name for the Section. This edit box has a 30-character limit. To enter a description, position the cursor in the edit box and enter your description.
Section Name	<Section N+1> (default) Any value you add.	Be as descriptive as you can, keeping in mind that you only have an 8-character limit. Develop a naming convention and be consistent throughout your projects.
Market	GBL (Default) [Market code]	Select the Market for which the Section is intended. For instance for the United States, pick USA and for Japan, pick JPN. If a particular Market is irrelevant to your batch program, leave the Market value as GBL. To select a Market code, position the cursor in the edit box, and select the appropriate option from the drop-down list.
Platform	Default DB2/MVS DB2/Unix Informix Oracle SQL Server Sybase	In some cases, you may be developing an application to run on a particular platform or RDBMS. This enables you to take advantage of any RDBMS-specific optimizations or to change your section logic because of RDBMS limitations. To select a Platform, position the cursor in the edit box, and select the appropriate option from the drop-down list.
Effective Date	<1900-01-01> (Default) <Appropriate Date...>	To make a particular Section Effective Dated (Active or Inactive based on a specified date), enter the target date in this edit box. To enter a date, position the cursor in the edit box, and either manually type in the date, or you can use the spin controls or the calendar drop-down control on the right to select a date.
Effective Status	Active (Default) Inactive	Use this control to specify whether a Section is active or not. Active refers to being enabled at runtime. To select an Effective Status, position the cursor in the edit box, and select the appropriate value from the drop-down list.
Section	Prepare Only	In the case of an abnormal termination of the

Control	Values	Details
Type	Critical Updates	<p>program, the value of this system field specifies whether or not you will need to Restart the Section.</p> <p>If a Section controls a procedure that, if not run to completion, could corrupt or desynchronize your data, choose Critical Updates.</p> <p>Otherwise, use the default value of Preparation Only.</p>
Auto. Commit	None (Default) After each Step	<p>Specify the Commit level for the Section. You can opt to have no Commit or you can have Application Engine Commit after the Step successfully completes. By clicking in the checkbox, you will toggle the value of this control.</p> <p>For more information on Commit, see Application Engine Commit.</p>
Access	Private (Default) Public	<p>This property controls the Call Section Action. You can specify that a Section can or can't be called from another program. To specify Private, deselect the Public checkbox. To specify Public, select the Public checkbox.</p>





Filtering Sections

Typically, you'll want the Section Filter settings to only allow unique Sections to appear in the view at a given time. However, suppose you update the Section Filter settings so the view does not filter by Platform yet you've defined multiple platform-specific definitions of a particular Section. In this situation, the view would display multiple Sections with the same Section name.

To help you identify particular definition objects in this case, additional information appears on the first row of Section definition object, following the short description text. The extra attributes that appear for identification are:

- Market
- Platform
- Effective Date

The following example reveals how these attributes might appear on the collapsed Section nodes.

 SectionA	SectionA description	SectionA.GBL.(base).2000-02-12
 SectionA	SectionA description	SectionA.GBL.(base).2000-06-01
 SectionA	SectionA description	SectionA.FRA.(base).1900-01-01
 SectionA	Section2 description	SectionA.USA.(base).1900-01-01

Collapsed Sections Revealing Identification Attributes

Execution Precedence

A Section is unique based on the program and Section names, and it is also unique based on its intended RDBMS platform and Effective date. Also, keep in mind, that in addition to these previous unique qualities you can specify for a Section, you can also create Market-specific Sections. When you execute an Application Engine program, it executes Sections based on the following order of precedence:

- If a Section for the current Market exists, execute it.
- If a Section for the current Platform, or RDBMS, exists, execute it.
- If a Section for the current Effective Date exists, execute it.

So, if a “unique” Section does not exist, Application Engine just executes the “default” Section.

For example, suppose you have two versions of a particular Section: SECT01 for Public Sector and SECT01 for Global use. If you request to run the Public Sector version of the program, Application Engine executes the Public Sector version of SECT01. If the program is running on Oracle, Application Engine then looks for an Oracle version of the SECT01 for Public Sector.

The default Market value is GBL for Global.









Inserting Sections

The following procedure outlines the basic steps you need to complete to insert and define a new Section within your program.

To insert a Section

1. Select **Insert, Section**, or select **Insert Section** from the pop-up menu.

The default name for a Section that you insert is Section N , where N is an incremented number that provides a unique name for each section object. Unless you rename your Sections according to a custom, naming scheme, the Sections you add will be named Section $N+1$, where n is the last Section you inserted. Consequently, you’ll end up with Section1, Section2, Section3, and so on.

		Section1	Section1 description	Section1.GBL(base).1900-01-01					
		Section2	Section2 description	Section2.GBL(base).1900-01-01					
		Section3	Section3 description	Section3.GBL(base).1900-01-01					
		Section4	Section4 description	Section4.GBL(base).1900-01-01					
			Market:	Platform:	Effective Date:	Effective Status:	Section Type:	Commit:	Access:
			GBL	(base)	01/01/1900	Active	Prepare Only	<input type="checkbox"/> None	<input type="checkbox"/> Private

Default Sequential Naming Scheme when Inserting Steps

Keep in mind that the designer inserts the new Section directly beneath all the subordinate

objects within the highlighted object's owning section. For instance, in the previous example, if Section2 were selected instead of Section3, then Section4 would be inserted *between* Section2 and Section3 rather than *after* Section3.

2. If you want to use a custom name for the Section, place your cursor in the Section name edit box, select the text to replace, and enter the desired name.

When adding custom Section names, keep the following in mind:

- Section names are limited to 8 characters.
 - Avoid special characters except for dashes (-) and underscores (_).
 - Be consistent with your naming conventions.
3. Place the cursor within the Section description edit box, and enter a description of your Section.

There is a 30-character limit. This description also appears with the section name and unique key attributes in the Program Flow view to help identify the Section.

4. From the **Market:** drop-down list, select the appropriate Market code for the Section.

The Market drop-down list is populated based on values stored in the Translate table. Your values may differ from those that appear in these examples.

5. Select the **Platform**.

From the drop-down list, select the platform for which this Section is intended. If it is not intended for a particular platform or RDBMS, leave the "default" value selected.

6. Enter the Effective Date and Effective Status.

Sections are effective dated. This means you can activate a particular section, using the **Effective Status** field, as of a specific date. This enables you to retain archived sections as you modify them instead of destroying them. If at a later date you decide to go back to using a previous version of a Section, you can simply deactivate the current section and reactive the previous one. When defining a new Section, you'll typically use the default settings for Effective Date and Effective Status, which are *1900-01-01* and *Active*, respectively.

To specify a particular date, place your cursor in the **Effective Date** edit box and either enter the numerals manually, use the calendar controls located on the right side of the edit box, or you can use the graphical calendar control, and click the appropriate date.

7. Choose the Section Type.

The **Section Type** option sets a value in the AERUNCONTROL table when the Section runs. And, in the case of an abend, the value of this system field specifies whether or not you will need to Restart the Section.



For more information see *Developing for Restart*.

8. Choose the Auto Commit value.

This specifies the commit capabilities for the Steps in the Section. If you specify no commits for a Section, then the commits are controlled by the Steps within the Section. If you specify at the Section level to commit after each Step, then the Steps are controlled by the Section setting.



Note. When an Application Engine program successfully completes its execution, there is an implicit commit performed.

9. In the **Access** checkbox, choose whether an external program can call the Section.

In a previous version of Application Engine, a program could call any Section defined in any other, external program. Now, you can selectively define whether a Section can be called from Sections defined in other programs.

The **Access** checkbox allows you to toggle between *Private* and *Public* access. Or, rather, you can toggle *Public* access on or off. The descriptions of each access type are as follows:

- **Unselected (off).** Indicates that a particular Section can only be called from “internal” Sections, or Sections defined within the same program. To restrict access to the current program only, make sure that *Public* is not selected.
 - **Selected (on).** Indicates that a Section can be called from both “internal” and “external” Sections. To enable Public Access, select the *Public* checkbox.
-



Note. The default Access is not *Public*.



For more information on calling Sections, see *Call Section Actions*.

10. Save your changes.

Locating Sections

When you have Application Engine programs calling sections from other programs or program libraries, it is helpful to be able to automatically determine the number of references to a particular section and to find the calling sections. And although we recommend that you keep

each program definition relatively small, when you do need to navigate within a large program, can require scrolling through numerous other sections.

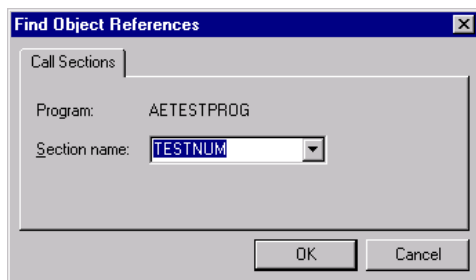
The following topics describe methods you can use to locate references to sections within an entire database as well as a method for easily navigating to particular section contained within a large program.

Finding Object References (Call Sections)

The following procedure describes how you generate a list of all the references to a particular section. The list only applies to Application Engine programs defined within a single database.

To locate object/section references

1. Open the program containing the shared or "called" section.
2. Select Edit, Find Object References.
3. The **Find Object References** dialog box appears.
4. On the **Call Sections** tab, select the appropriate section from the **Section name** drop-down list, or enter the name in directly.



Find Object References dialog box

By default, the current program name appears as the Program value.

5. Click **OK**.
6. In the output window, view the generated list.

The output window lists the programs and section that call a particular program, and it also shows the total references made to a particular section. Call Sections within the current program appear first in the list.

```
Searching for references to Program 'AETESTPROG' Section 'PCODE'...
Found: Program 'AETESTPROG' Section 'MAIN' Step 'PCODE'.
Found: Program 'GREGAE' Section 'Section1' Step 'Step01'.
Found 2 reference(s) to Program 'AETESTPROG' Section 'PCODE'.
```

Find Object References output

Double-click an item in the output window list to automatically navigate the definition view to that calling section.

Finding Sections within the current Program

Within large and more complicated Application Engine programs, such as those upgraded from previous release, it is not uncommon to have over a hundred sections. In this circumstance, finding a particular section by scrolling and clicking can be a time consuming and tedious task. Rather than scrolling through a large program use the Go To Section feature.



This feature only applies to the current program. For example, you do not use Go To Section to navigate to a section in another program.

To automatically navigate to a selected section

1. Select Edit, Go To Section.

The **Find Object References** tab appears.

2. On the **Go To Section** tab, select the appropriate section from the **Section name** drop-down list or enter the name of the section.
3. Click **OK**.

The Definition View scrolls to the first occurrence of the section with the name you selected.

Steps

Once you've defined a Section, you need to define the Steps for that Section. Each Step contains Actions, which execute SQL statements, program flow control statements, message log statements, or PeopleCode.

A Step represents the smallest unit of work that can be committed in a program. When you first create a program, you have a default MAIN Section and Step, initially named Step01. All programs require at least one Section, all Sections should contain at least one Step, and all Steps should contain at least one Action.

Step Properties

The next level of properties that you can define applies to the Application Engine Step. As with the Section properties, you need to have the Definition view active to modify any properties.

To modify or view the Step properties you need to expand the Section node so the Step nodes it contains are visible. The following is an example of the default values that appear on a Step when you first insert it.

Step Properties on the Step Node

As with the controls on the Section node, you merely need to position the cursor inside an edit box so that it becomes active. Then you can either enter text manually or use the drop-down list, or the appropriate control, to select the desired options.

The following table contains the properties available for Steps.

Control	Values	Details
Step Name	Step(n+1) (Default) Or, any name you provide.	According to your Step-naming convention, uniquely identify each Step. This edit box has an 8-character limit.
Step Description	<Step Name> description (Default) Or, any description you provide.	This edit box has a 30-character limit. Here, you may want to briefly describe the Step's purpose.
Commit	Default (Default) After Step Later	<p>This is where you specify the Commit level for the Step. The following briefly describes the Commit options at the Step level:</p> <p>Default. You can opt to select <i>Default</i>, which means the Step will inherit what ever Commit level you have specified for the Section in which the Step resides.</p> <p>Later. By selecting <i>Later</i>, you postpone the commit until a subsequent Commit occurs, as specified by developers. Again, here you can override the Section-level commit, if it happened to be set to <i>After Step</i>.</p> <p>After Step. Selecting <i>After Step</i> is useful if you have a Commit level of <i>None</i> specified at the Section level. This way, you can override the Section-level commit, and commit a specific Step within a Section with no other commits.</p> <hr/> <p>For more information on Commit, see Application Engine, "Advanced Development".</p> <hr/>
Frequency	Blank (Default) Any number you add.	The Frequency control only becomes enabled when a Step contains one of the following Actions: Do While, Do Select, or Do Until. This control will only accept numeric values. The value indicates the frequency with which Application Engine should commit. If non-zero,

Control	Values	Details
		Application Engine commits every <i>N</i> iterations, and then again after the last iteration.
On Error	Abort (Default) Ignore Suppress	<p>Here you can specify how Application Engine should respond to an error at the Step level. The On Error routine behaves the same for both SQL and PeopleCode Actions. The program only aborts on errors, not warnings.</p> <p>Abort. The application will terminate with an error message.</p> <p>Ignore. The program continues but logs an error message.</p> <p>Suppress. The program will continue and present no error message.</p>
(Active) Status	Active (Default) Inactive	<p>Here you can activate a Step or make it inactive. If the Step is currently applicable to your program (and working) you'll probably want to keep it <i>Active</i>. However, if you're still developing or testing the Step you may want to make it <i>Inactive</i>. The checkbox toggles the status of a Step. An inactive step does not have the checkbox selected.</p>

Inserting Steps

The following procedure outlines the steps you need to complete when you insert a Step into your program.

To insert a Step

1. Highlight the Section or Step that you want to immediately precede the new Step.

For example, if you want the new Step to be the *first* Step in the Section, select the Section node. Otherwise, select the existing Step that you want the new Step to follow. The insert always occurs *after* the currently selected node.

2. Select Insert, Step/Action.

By default, the Steps will be numbered as *StepN+1* beginning with *Step01*.



Note. The designer will continue to increment the Step Name until it has a unique Step Name within a particular Section. If the designer is unable to create a unique name after 50 attempts, a new Step will not be inserted. Consequently, you should devise your own naming convention to define more descriptive Step names.

3. Specify a Step name.

If you want to rename the Step name, position the cursor in the Step name edit box and enter a custom name. We only recommend accepting the default name for building quick, simple programs and for training purposes.

4. To the right of the Step name edit box, enter a description of the Step.

This description will appear in the Program Flow view. The control imposes a 30-character limit.

5. From the **Commit** drop-down list select the appropriate option.

This is where you can override the “default” Commit level, or the Commit level you set at the Section level. Remember that the Section-level Commit applies to all Steps within the Section unless you override it for a particular Step.



Note. Often you will see the word “checkpoint” used in discussions of Commit. A “checkpoint” occurs when Application Engine updates the State Records in the database prior to performing a commit. Application Engine always checkpoints prior to a commit so that a restart is possible.

At the Step level, you have the following Commit options:

- **After Step.** Commit after successfully finishing the execution of this Step. There is an implied “this” in between the "After" and "Step." If your Section level Commit is None, you can set the Commit level of a particular Step.
 - **Later.** A Commit may occur, just not after *this* Step. For instance, if at the Section level you have specified to commit after every Step, you can opt to *not* commit after a particular Step. Perhaps you only want to commit after the *next* Step completes.
 - **Default.** Commit based on the Commit level set at the Section level.
6. From the Error drop-down list, select the option that reflects how you want Application Engine to handle possible errors.

Your options for error handling are:

- **Abort.** If any error occurs, the system performs database rollback and processing halts.
 - **Ignore.** If an error occurs, details will be written to the Message Log but processing will continue.
 - **Suppress.** Processing continues and details regarding the error are not written to the Message Log. Perhaps you know you’ll be *expecting* “Duplicate Insert” errors. With this option you can continue without filling up the Message Log unnecessarily.
7. Specify the Active status of the Step.

Use the Active checkbox to toggle the Step status to Active and Inactive. The default is

Active, which means the Step will be executed. An *Inactive* Step is ignored at runtime. This functionality is equivalent to commenting out a step in SQR or COBOL. This option is convenient when you want to temporarily disable a Step from running rather than having to delete it and add it again later.

To make a Step Inactive, make sure that the *Active* checkbox is *not* checked.

Actions

There are eight types of Actions that you can include within a Step, and a Step can contain multiple Actions. Which Action(s) you choose to define for a Step depends on the type of results that your program requires at each stage of execution.

The only mutually exclusive Actions within a single Step are Call Section and SQL Statement; in other words, you can't add a Call Section Action to a Step that already contains a SQL Statement Action and visa versa. Furthermore, you can only include one of each Action type within a single Step. Since there are eight types of Actions, and two are mutually exclusive, the maximum number of Actions a single Step can contain is seven.

Action Properties


Just as Steps appear beneath the Section to which they belong, Actions for a Step appear beneath the Step to which they belong. And as with the previous definition objects discussed, you can specify object-specific properties for each Action. To modify Action properties, you need to have the Definition view active.


With Sections and Steps, every other Section or Step has a consistent set of properties or attributes that you can specify for the object. For instance, each Step has a Commit, On Error, and Active property, and, for each Section, you can define the Market, Platform, Effective Date, and so on.

For Actions it's different. Because there are a variety of Actions that you can include within a Step, there are different sets of properties specific to a particular Action type. Depending on what Action type you choose, the properties that appear will change.

For example, with a SQL Action, you can specify whether or not the statement will take advantage of the ReUse feature. The ReUse feature doesn't apply to a PeopleCode Action, so with a PeopleCode Action you will instead need to specify how Application Engine should respond if your PeopleCode program provides a False return.

The following example depicts how for different Action types you can select Action-specific properties.

 Do Select	<i>Do Select description</i>
ReUse Statement: Do Select Type:	
No	Select/Fetch

 Call Section	<i>Call Section description</i>
Section Name: Program ID:	
STATS	AETESTPROG <input type="checkbox"/> Dynamic

Actions and Associated Properties

PeopleCode and all SQL-related Action types invoke the related PeopleTools Editor to define or maintain the related text. The following table describes the properties you can select for each Action type. The Actions are arranged according to properties that apply to them.

Action Type	Object Type	Configurable Properties	Editor
Do When Do While Do Until	SQL Select (Iterative)	<p>ReUse Statement: Selecting this means the database engine only needs to compile the SQL once, which reduces SQL overhead. Choices are Yes, No, and Bulk Insert.</p> <hr/> <p>For more information on ReUse and Bulk Insert, see Application Engine, “Advanced Development” .</p> <hr/>	SQL Editor
Do Select	SQL Select	<p>ReUse Statement: (See Do When, Do While, and Do Until.)</p> <p>Do Select Type: You need to specify one of the following: Select/Fetch, Reselect, or Restartable.</p> <hr/> <p>For more information on Do Selects, see DO Select .</p> <hr/>	SQL Editor
PeopleCode	PeopleCode	<p>On Return: If your PeopleCode program provides a false result, you can have Application Engine respond by doing one of the following: Abort, Break, or Skip Step.</p> <hr/> <p>For more information on using PeopleCode with Application Engine, see PeopleCode Actions.</p> <hr/>	PeopleCode Editor
SQL	SQL Statement	<p>Reuse Statement: (See Do When, Do While, and Do Until.)</p> <p>No Rows: If your SQL statement doesn’t affect any rows, you can specify</p>	SQL Editor

Action Type	Object Type	Configurable Properties	Editor
		that Application Engine respond in one of the following ways: Abort, Section Break, Continue, or Skip Step.	
Call Section	Application Engine Section	<p>Section Name: If you are calling another Section, you need to specify the appropriate Section here. Only the names of Public Sections defined in the program identified in Program ID will appear in the Section Name drop-down list.</p> <p>Program ID: Since you can call Sections in the current program or Sections that exist within other programs, you need to specify the Program ID, or Program Name, of the program containing the Section you intend to call. The drop-down list contains all of the program definitions that currently exist.</p>	N/A
Log Message	Message Log	<p>Message Set: Identifies a message set within the Message Catalog.</p> <p>Number: Identifies a particular message within a Message Set.</p> <p>Parameters: A list of comma-delimited values to substitute for %1, %2, and so on markers in the message text.</p>	N/A

Keep the following rules in mind when inserting Actions—the system won't allow the following:

- You can't have more than one Action of a specific type within the *same* Step.
- You can't have a SQL Action and a Call Section Action within the *same* Step.



For more information on Action types, see Actions.

Inserting Actions

In this topic, we describe the general procedure that you will follow to insert an Action within a Step. Keep in mind that there are multiple Action Types, and each type offers a unique set of properties for you to specify. So, look to this procedure to show you the basics of inserting Actions. For specific properties related to a given Action, see the following topics, which contain details specific to each Action type.

To insert an Action

1. Highlight the Step in which you want to insert the Action.
2. Insert the Action.

You do this using one of the following methods:

- Select Insert, Step/Action.
 - Right-click on the Step and select **Insert Step/Action** from the pop-up menu.
3. Select the Action type from the drop-down list.

The default Action type for the initial Action inserted into a Step is SQL. If you wish to define an Action type other than SQL, select the appropriate type from the drop-down list or manually enter the name.

Notice that when you select different Action types, the properties that you can define change to reflect the appropriate properties of the Action you selected. For example, with a SQL Action, you can define the ReUse Statement and No Rows properties, but if you select PeopleCode, you can only define the On False property.

4. Enter a description of the Action in the description edit box.

This is the description that appears in the Program Flow view. This edit box has a 30-character limit for your descriptions, so plan accordingly. Longer blocks of descriptive text should be included in the Comments section, which is available in both the Definition view and the Program Flow view.

5. Specify the appropriate properties for the Action you selected.

For example, if it's a Call Section Action, specify the location of the Section you're calling, and if it's a Do When Action, specify the ReUse Statement properties.



For more information on the specific properties for an Action, see Action Properties.

The following sections describe in more detail the Action types and the properties that are specific to each Action type.

SQL Actions

This is the default Action type for the first Action within a given step and any new Actions if you have not already defined an Action of this type. Use this Action if you want to perform the following SQL commands on multiple rows:

- UPDATE

- INSERT
- DELETE
- SELECT



Note. Before you insert SQL (View, SQL) into a SQL Action within a new Application Engine program, you need to have previously saved the program. This is required because the program name you use to save this definition with is used to relate your program with the SQL objects you are about to create. The same is true for inserting new PeopleCode.

With a SQL Action, you use the SQL Editor to create and modify your SQL statement. For example,

```
%Select (AF_PERFM_AET.PREV_ASOF_DT)

SELECT %DateOut (ASOF_DT)

FROM PS_AF_FCST_SCHT%Bind (EPM_CORE_AET.TABLE_APPEND,NOQUOTES)

WHERE AFDEFN_ID = %Bind (AF_CORE_AET.AFDEFN_ID)

AND ASOF_DT = (

SELECT MAX (ASOF_DT)

FROM PS_AF_FCST_SCHT%Bind (EPM_CORE_AET.TABLE_APPEND,NOQUOTES)

WHERE AFDEFN_ID = %Bind (AF_CORE_AET.AFDEFN_ID)

AND ASOF_DT < %Bind (AF_PERFM_AET.ASOF_DT))
```



Note. If you intend to include multiple SQL statements within a single Application Engine Action you should use the meta-SQL construct %EXECUTE. The previous sample SQL statement sample contains bind variables from a previous Application Engine Action.



For more information on %EXECUTE, see Application Engine Meta-SQL.

The following topics describe the properties that you can specify for SQL Actions.

ReUse Statement

ReUse is an option you can enable to optimize the SQL components of your batch program. ReUse converts any %BIND references to State record fields into real bind variables (:1, :2, and so on), allowing the Application Engine runtime process to compile the statement once, dedicate

a cursor, and then re-execute it with new data as often as your program requires. When you are using SQL to process a large volume of rows, one at a time, inside a fetch loop, compiling each statement you issue can be a considerable performance issue. ReUse is a way to combat potential performance decreases.

You have the following options when it comes to applying ReUse to your SQL Actions.

- **Bulk Insert.** When used in conjunction with statements like INSERT INTO tablename (field1, field2...) VALUES (%BIND(ref1), %BIND(ref2), Bulk Insert offers the most powerful degree of performance enhancements related to ReUse. This option turns on ReUse, and, in addition, it holds all the data in a buffer and only performs an insert after a large volume of rows has gathered in the buffer. The number of rows allowed to gather in the buffer depends on your database platform. Keep in mind that storing data in the buffers is only applicable if you've selected Bulk Insert *and* the SQL is an INSERT statement. For statements other than INSERT/...VALUES, the Bulk Insert option is ignored.
- **No.** Select this option to disable ReUse. With ReUse off, the Application Engine runtime process recompiles the SQL statement every time the loop executes. By default, ReUse will be disabled.
- **Yes.** Select this option to enable basic ReUse functionality.

ReUse can offer significant performance gains, but we do not intend it to be used for *all* SQL statements. For instance, here are some basic guidelines:

- Don't use ReUse if you use %BIND variables to build parts of the SQL statement, unless the STATIC option is used on the %BIND.
- Don't use ReUse if you have %BIND variables in the field list of a SELECT statement unless the value is constant, so you can use the STATIC modifier.
- You can use %SQL as long as the SQL it references conforms to the rules of %BIND.



Note. You can have Application Engine recompile a reused statement by using the %ClearCursor function.



For more information about ReUse, see Re-Using Statements, and For more information about %BIND and %ClearCursor, see Application Engine Meta-SQL.

No Rows

In the event that the SQL (INSERT, UPDATE, and DELETE) associated with the SQL Action does not return any rows, you need to specify what your Application Engine program should do.

For example, you could use this in a case where you INSERT into a temporary table, and then you intend to perform further operations on the inserted rows (provided some rows meet the criteria). If the initial INSERT...SELECT provides no rows, you could save the program from

having to re-SELECT on the temporary table prior to executing another operation, or you could also prevent the program from performing set operations on the table when there won't be any qualifying rows.

The following list contains the options that you have when no rows are returned.

- **Abort.** The program terminates.
- **Section Break.** Application Engine exits the current Section immediately, and control returns to the calling Step.
- **Continue.** The program continues processing.
- **Skip Step.** Application Engine exits the current Step immediately and moves on to the next Step. When using Skip Step keep the following in mind:
 - Application Engine ignores the commit for the current step at runtime.
 - If the current Step contains only one Action, only use Skip Step to bypass the commit.



Note. Using No Rows in conjunction with a Truncate Table operation is unreliable. Some database platforms report "zero rows affected" for truncates, regardless of how many rows were in the table.

Program Flow Actions

There are four types of Application Engine Actions that, although distinct from the others, can be grouped together under the category of Program Flow Actions. This category includes the Actions, which appear in the following list:

- Do When
- Do While
- Do Select
- Do Until

Use the Program Flow Actions to control the execution of your program. With these Action types you can control the execution of subsequent Sections, Action(s), or SQL statements depending on the results of a “Do” SQL statement in the form of a SELECT. If you were coding in COBOL, you would perform similar actions using the IF and WHILE functions.

Any of the Program Flow Actions can control the execution of a Section, a SQL statement, a PeopleCode program, or a Log Message. In previous versions of PeopleTools, you needed to specify the Section affected by the Program Flow SELECT. So, the Program Flow Actions can control the execution of a Section or a single Action. For example, a Do Select can execute a SQL statement for each row returned by the included SELECT.

The following topics describe each Program Flow Action and the options that are specific to the program flow Action type.

DO When

The DO When Action is a SELECT statement that allows subsequent actions to be executed if any rows of data are returned.

This Action is similar to a COBOL “IF” statement. A DO When statement runs *before* any other actions in the Step. If the DO When statement returns any rows, the next Action will be executed. If the Do When conditions are not met, the remaining Actions within that Step are not executed. Your program executes a DO When Action only once when the owning Step executes.

The only property that you can specify for the Do When Action is the ReUse Statement property, which applies to all SQL-based Actions.



For more information on selecting the correct level of ReUse, see ReUse Statement.

DO While

The DO While Action is a SELECT statement that, if present, runs *before* subsequent Actions of the Step. If the Do While does not return any rows of data, the Action terminates. The Do While is identical to the COBOL “WHILE” function. In other words, the subsequent Actions within the Step are executed in a loop as long as at least one row is returned by the SELECT statement for the DO While Action.

In short, if the Do While does not return any rows, the Step is complete.

The only property that you can specify for the Do While Action is the ReUse Statement property, which applies to all SQL-based Actions.



For more information on selecting the correct level of ReUse, see ReUse Statement.

DO Select

The DO Select Action is a SELECT statement that executes subsequent Actions once for every row of data that the Do Select returns. For instance, a DO Select can execute a SQL statement for each row returned from the SELECT statement. The subsequent Actions within the Step are executed in a loop based on the results of the SELECT statement. The type of the DO Select determines the specific looping rules.

When you insert a Do Select Action, you need to specify the Action properties described in the following topics.

ReUse Statement

As with any SQL-based Action, with a Do Select you have the option of specifying the ReUse Statement level.



For more information on ReUse, it is described in a previous topic: ReUse Statement.

Do Select Type

When you add a Do Select Action, you can specify the ReUse Statement option available for all SQL-based Actions, but you also need to specify the Do Select Type. The following list describes the three types of Do Selects you can choose:

- **Select/Fetch.** With this option, Application Engine opens a cursor for the DO Select, then, within that cursor, Application Engine performs a Fetch for each iteration of the loop to get each row from the SELECT. When a FETCH results in an “end of table”, the looping is complete. You can’t restart this type of SELECT statement since Application Engine does not perform a checkpoint or a commit within the Step containing this Action while Select/Fetch is running. Ultimately, your program ignores the commit settings at runtime until the outermost Select/Fetch completes.
- **Re-Select.** For each iteration of the loop, Application Engine opens a cursor and fetches the first row. This means that, with Re-Select, your program processes the “first row” returned from the Select statement. It also means that the cursor gets re-opened for each iteration of the loop. With this type of Fetch, you will typically want some aspect of the loop to eventually cause the Select to return no rows. Otherwise, there is no mechanism in place by which to exit the loop. This type of DO Select *is* restartable.
- **Restartable.** This option is similar to the Select/Fetch in that Application Engine opens the cursor associated with the DO Select once, and then it performs a Fetch on each iteration of the loop to get each row from the SELECT. However, unlike the Select/Fetch option, you can restart this type of Select and Fetch because Application Engine performs a checkpoint in the middle of the Step. Keep in mind that Application Engine will only *treat* this loop as if it is restartable. Developers need to make sure that the SQL they include within this Action is such that, upon restart, it recognizes where the previous run failed and where to restart processing. There are various techniques you can use to achieve this, such as employing a “processed” switch, or to base the next Select on the key.

Special Concerns with DO Select

Application Engine does not commit a Step containing a DO Select with the *Select/Fetch* option enabled until the entire Step completes successfully. This means that with this type of Step no commits (or checkpoints) occur until the Step completely finishes executing, regardless of what other options you have selected.

For example, suppose at the Step level you specified to COMMIT every 100 iterations of the Step. One of the Actions of this Step is a DO Select with Select/Fetch chosen. Because Application Engine will not checkpoint or commit while the DO Select is active, the transaction

performed by the Actions within a Step will not be committed until the entire Step completes successfully. Again, this is also true if any Sections are called from inside the loop.



For more information about Restart, see Developing for Restart.

DO Until

A Do Until Action is a SELECT statement that runs *after* each Action when a Step completes. If the SELECT returns any rows of data, the Step terminates.

Use a Do Until if you want the “processing actions” to execute at least once, and to execute over and over until a certain condition is true, such as until a SELECT returns some rows.

You can also use a Do Until to stop a Do Select prematurely. For example, if the SELECT for the Do Until does not return any rows, then the Actions in the Step are repeated (except if a Do When appears in the Step). Normally, a DO Select continues until no rows are returned. If any rows of data are returned, the DO Select stops and the Step is not repeated.

The only property that you can specify for the Do Until Action is the ReUse Statement property, which applies to all SQL-based Actions.



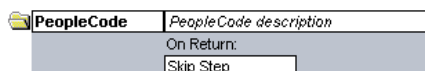
For more information on selecting the correct level of ReUse, see ReUse Statement.

PeopleCode Actions

Use this Action type to insert a PeopleCode program within your Application Engine program. You can invoke the PeopleCode Editor right from the designer interface to code your PeopleCode programs.

PeopleCode Actions allow you to take advantage of IF-THEN logic, messaging, File Layouts, Components, and various other features all from within your Application Engine programs. Although we encourage you to use PeopleCode in your Application Engine programs, there are some “recommended” uses that you’ll want to be aware of prior to any serious design and code work.

With a PeopleCode Action, there is only one property that you can specify: **On Return**.



PeopleCode Action Properties

Use the On Return value to determine how your Application Engine program will react based on the return of your PeopleCode program. The On Return setting takes effect if your PeopleCode

program issues a “return 1” or “exit 1.” You can use the TRUE keyword in place of a non-zero numeric return.

The following list contains the valid options for this On Return:

- **Abort.** The program issues an error and exits immediately.
- **Break.** The program exits the current Step and Section, and control returns to the calling Step.
- **Skip Step.** The program exits the current Step, and continues processing at the next Step in the Section. If this is the last Step in the Section, the calling Step resumes control of the processing.

The following is an example of some PeopleCode one might add to a program.

```
If All (FSI_CORE_AET.PF_CONSTRAINT_CODE) Then  
    Exit True;  
Else  
    Exit False;  
End-If;
```




For more information see Notes on Various PeopleCode Objects and Functions and, most importantly, PeopleCode Developer's Guide.

Call Section Actions

Use the Call Section Action to call another Section defined in an Application Engine program. You can call a (local) Section defined within your current program, otherwise Section MAIN would be the only Section executed. But, also you can make external calls to a Section defined in another Application Engine program.

The external Section you intend to call must have its Access property set to *Public*. If a Section's Access property is set to *Private*, that Section can only be called from within the same program. By default, a Section's Access property is *Private*. If you attempt to make a call to a Section that does not allow external calls, you receive an error message.

Call Section is the only supported way to call Application Engine programs or Sections from other Application Engine programs or Sections. You can't call Application Engine programs or Sections from PeopleCode Actions.

 Call Section	<i>Call Section description</i>
Section Name:	Program ID:
ENGMSG	AF_COMMON
<input type="checkbox"/> Dynamic	



Note. You can only call programs that reside within the same database as the “calling” program.

The following topics describe the properties you specify for a Call Section Action as well as some items to consider when using the Call Section functionality.

Section Name

When you use the Call Section Action, whether or not the Section is defined in the current program or an external program, you still need to specify the appropriate Section name. The Section names that appear in the **Section Name** drop-down list are those that are defined in the program that appears in the **Program ID** edit box. So, if you want to call a Section that is defined in an external program, always select the program name in the Program ID edit box prior to selecting the Section name.

You can also use the Call Section Action to call an entire external program. First select the Program ID, then select Section MAIN. At runtime, this call executes the entire program defined by the value in Program ID.



Note. Note. Application Designer does not prevent you from calling the Main Section of the current program or the current Section. For instance, Section1 can contain a Step that has a local Call Section reference for Section1. This enables recursive calls, and should therefore be used with caution.

Program ID

Since you can call Sections defined in the current program or Sections that are defined within external programs, you need to first specify the Program ID (Program name) of the program containing the Section you intend to call.

The default value is (*current*), which indicates local Section calls. If you intend to call a Section defined in another program, you'll have make sure that you first select the appropriate external program from the drop-down list. The drop-down list contains the names of all program definitions that currently exist in the database.

Dynamic

Rather than calling one specific Section, you can take advantage of the AE_APPLID and AE_SECTION fields in the State record to execute different Sections depending on the conditions a program encounters during runtime. This is referred to as a Dynamic Call.

These two fields must be defined on the default State Record for the program. If AE_APPLID is not present or is blank (at run time), the current program will be substituted for the AE_APPLID value. If AE_SECTION is not present or is blank, an error will occur.

When issuing a Dynamic Call, both the Section and the Program ID must be dynamically set. For instance, you can't define a value for the Program ID and leave the Section edit box blank.

You enable a Dynamic Call by first having your program store different Section names in the AE_SECTION field, and different program names in AE_APPLID field. The values you insert in these fields are normally based on various conditions met within your program. You then create a Call Section Action that calls the Section name defined in the State record field by selecting the **Dynamic** checkbox.

Selecting Dynamic automatically populates the AE_SECTION field with the symbolic value %Section, and the Program ID field with the symbolic value %AEAPPLID. At runtime, the program calls the Section name stored in AE_SECTION that belongs to the program name defined by AE_APPLID.

Program Properties

When you call a Section defined in an external program, the current program (the program containing the defined Call Section) defines the properties that apply to the running process. Suppose tracing is enabled for the current program, but tracing is disabled for the called program Section. In this case, the called program has the trace option enabled at runtime because it inherits the calling program's properties.

For example, if Program A calls Program B *and* Program B calls Program C, then the properties of A apply to both programs B and C. The calling program always controls the properties for the called program. In this case, Program A controls the properties for Program B, and because Program B inherits the properties of Program A, when Program B calls Program C, Program A's properties *also* apply to Program C.



Note. Although program properties are inherited, note that State Records do not follow this inheritance model.

Sharing State Records

When you call a program from another program, the called program's default State Record becomes active until processing returns to the initial program. However, all of the State Records associated with both programs will be available. State Records that are common between the two programs "share" values.


This means that if you need to communicate between the two programs, as in share %BIND variables, you need to define the same State Record(s) in both programs.



For more information on State Records, see State Records.

Log Message Actions

Use this type of Action to write a message to the Message Log. The Message Log refers to the PeopleTools table PS_MESSAGE_LOG where execution messages reside. Any substitution parameters are written to PS_MESSAGE_LOGPARM.

 Log Message	Log Message description		
	Message Set:	Number:	Parameters:
	10662	278	%BIND(AF_CORE_AET.AFDEFN_ID)

You can use the Log Message Action to insert any type of messages you need. Typically, Log Message writes error messages to the Message Log, but you could also write informational or status messages as well.



Note. You can also use MessageBox PeopleCode to populate PS_MESSAGE_LOG instead of using the Log Message Action. This allows you to easily record errors encountered within Application Engine PeopleCode programs.

Message Set and **Number** identify the specific message defined in the Message Catalog, and the **Parameters** are a comma-delimited list of values to substitute for the message variables (%1, %2, and so on) in the message text. These parameters can be hard-coded values or %Bind references. The information specified is inserted in the PS_MESSAGE_LOG at runtime, and any %Bind values are replaced by the current State Record field values. You can then view the logged messages from the Process Monitor page.



Note. Both message set and number can be a hard-coded numeric value or a "[record.]field" reference for dynamic message logging (record is optional; if missing, the default State Record is assumed).

For example:

Message set 1012, number 10 = "The total number of %1 rows exceeds the control count value, %2."

Parameters:

Invoice, %Bind(CONTROL_CNT)

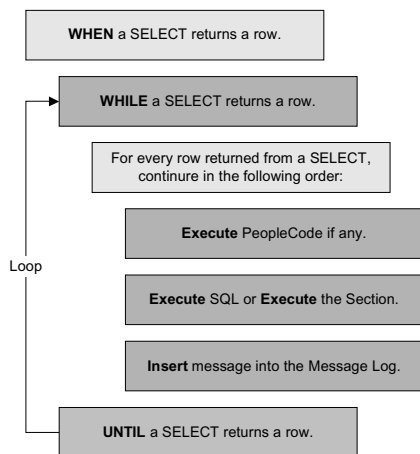
Suppose you run this program with the CONTROL_CNT field value of 120. When the process completes, the following message would be included on the Process Details dialog in Process Monitor:

The total number of Invoice rows exceeds the control count value, 120.

Action Execution Hierarchy

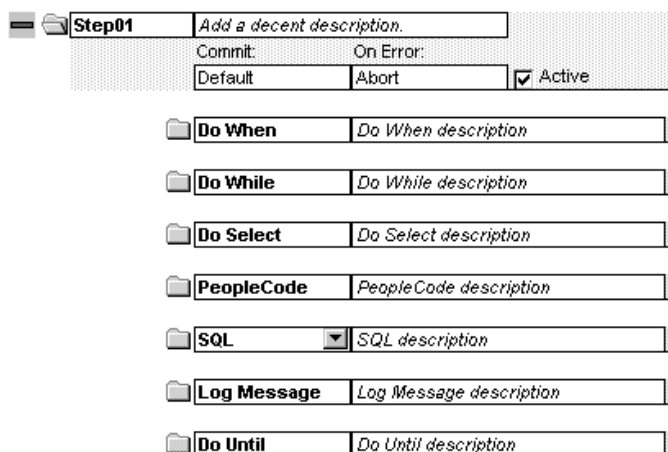
The support for multiple Action types requires that they execute in a consistent and reliable manner, and thereby the order in which Actions execute is significant. At runtime, Actions defined for a given Step are evaluated based on their Action type. All of the Actions types exist within a strict hierarchy of execution. For example, if both a Do When and PeopleCode Action exist within a given Step, Application Engine always executes the Do When first.

The following example showing the execution hierarchy for Actions depicts the sequence and level of execution for each type of Action:



Action Execution Hierarchy

Notice that as you add Actions to a Step in the Definition view, the Actions are initially inserted after the selected definition object (the owning Step or a prior Action). However, following a Save request or a Refresh of the view, the designer reorders all Actions to match the execution hierarchy. This feature helps you visualize the sequence in which each Step of your program logic executes.



Actions in Execution Order in Designer View



Note. In the Previous example, the SQL Action and a Call Section Action are interchangeable, in that one or the other would fill that position in the hierarchy. Only one of these two Action types can appear within a Step as they are mutually exclusive.

CHAPTER 4

Advanced Development

For the most part, you'll develop your Application Engine programs in the Application Engine Designer. However, creating programs and adding Sections, Steps, and Actions doesn't make up all of what's involved in the Application Engine Development process. In this section, we'll cover the aspects of Application Engine development that don't necessarily relate to the Application Designer, such as meta-SQL, the Application Engine Debugger, and Set Processing.

State Records

State Records are a key component for an Application Engine program. This section provides information that you should be aware of prior to constructing and planning the use of your State Records.

Overview

The Application Engine State Record is the method by which you allocate variables for your Application Engine program, and it is also the method by which Section, Steps, and Actions pass values to subsequent program steps. You can think of State Records in Application Engine as global variables without scope limiting rules.

You can have any number of State Records associated with a particular Application Engine program. However, only one record can be the default State Record. You can specify both work (derived) and "physical" (SQL table) record to be used as State Records. The only difference is that derived State Records cannot have their values saved to the database at commit time, and so the values would be lost during a restart. Because of this issue, Application Engine will erase the contents of derived State Records at commit time if Restart is enabled for the current process.

An Application Engine State Record must have `PROCESS_INSTANCE` defined as the first field and the *only* key field. And, so that the system recognizes the record as a State Record, all State Record names must end with the `_AET` identifier.

Not all the database columns referenced in your program need to appear in the State Record, just the columns that must be selected into memory, so that the values can be referenced in a subsequent program action. You may also want to include additional fields to hold pieces of dynamic SQL, to use as temporary flags, and so on.

Long Fields are supported and this is not the case in COBOL or SQR. However, PeopleSoft only allows one long per State Record. You set a maximum size for the field in the Application Designer, and when doing so, make sure that the data space is compatible with the size of the field that you set.

The following exhibit shows a sample State Record. Notice the _AET naming convention.

AE_TESTAPPL_AET (Record)							
Record Fields		Record Type					
Num	Field Name	Type	Len	Format	H	Short Name	Long Name
1	PROCESS_INSTANCE	Nbr	10			Instance	Process Instance
2	AE_PRODUCT	Char	2	Upper		Product	Product
3	AE_APPL_ID	Char	8	Upper		Appl ID	Application
4	AE_STEP	Char	8	Upper		Step	Name
5	AE_SECTION	Char	8	Upper		Section	Section
6	AE_INT_15	Nbr	15				Number Work Field
7	AE_INT_14	Nbr	14				Number Work Field
8	AE_INT_13	Nbr	13				Number Work Field
9	AE_INT_12	Nbr	12				Number Work Field
10	AE_INT_11	Nbr	11				Number Work Field
11	AE_INT_10	Nbr	10				Number Work Field
12	AE_INT_9	Nbr	9				Number Work Field
13	AE_INT_8	Nbr	8				Number Work Field
14	AE_INT_7	Nbr	7				Number Work Field
15	AE_INT_6	Nbr	6				Number Work Field
16	AE_INT_5	Nbr	5				Number Work Field
17	AE_INT_4	Nbr	4				Number Work Field
18	AE_INT_3	Nbr	3				Number Work Field
19	AE_INT_2	Nbr	2				Number Work Field
20	AE_INT_1	Nbr	1			Work Field 1	Number Work Field

Sample State Record

During batch processing (as opposed to online invocation using CallAppEngine in PeopleCode), Application Engine automatically performs all State Record UPDATES. When a program starts, it INSERTS a row into the State Record that corresponds to the Process Instance assigned to that program run. Application Engine updates the record whenever a COMMIT occurs. When Restart is enabled and a commit occurs, all State Records that have been updated in memory will be written to the database, except for derived State Records, which are instead initialized.

Then, after the program completes successfully, Application Engine deletes the corresponding row in the State Record. There is only one row in the State Record for each Process Instance, or program. Multiple programs can use the same State Record, and each program has it's own row based on the unique Process Instance key.

To set values in the State Record, you use the %SELECT construct in a SQL statement or write PeopleCode that references the State Field with the standard "record.field" notation. To reference fields in the State Record you use the %BIND construct.



For more information on %SELECT and %BIND, see the Meta-SQL section Application Engine Meta-SQL.



Note. When a State Record is a work record, no database UPDATES can be performed on the record. Consequently, if your program is restartable, your code must take into account that this “memory” will be lost if the program s. In fact, Application Engine automatically re-initializes fields on all work records after each commit.

Sharing State Records

The “scope” of the State Records will be global in that they are not only designed to service multiple Sections, they can also be used by multiple programs. When you call a Section in another program, any additional State Records defined for that program (as in State Records that are not already in use by the calling program) will be initialized, even if the program has been called previously during the run. But State Records that are common to both programs will retain their current values.

To reference variables that exist within a State Record, you use

```
%BIND(fieldname)
```

Unless a specific record name is specified preceding the fieldname, %BIND references the default State Record. To reference a State Record other than the default, you need to use

```
%BIND(recordname.fieldname)
```

In the case of a called program or Section, if the called program has its own default State Record defined, then Application Engine uses that default State Record to resolve the %BIND(fieldname). Otherwise, the called program will inherit the calling programs default State Record. In theory, the called program does not require a State Record if all the fields it needs for processing exist on the calling program’s State Record.



For more information on the %BIND construct, see %Bind.

For those State Records that are shared between programs (during an “external” Call Section), any changes made by the called program will remain when control returns to the calling program. This means that any subsequent actions in the calling program can access residual values left in the common State Record(s) by the called program. This can be useful at times to return output values or status to the calling program, yet it can also cause unforeseen errors.

Generally, a called program should not share State Records with the caller unless you need to pass parameters between them. Most programs will have their own set of State Records unless a program calls another program that requires specific input or output variables. In that case, you will need to include the State Record of the called program into the calling program’s State Record list, and make sure to set the input values before issuing the Call Section. If you are worried about sharing a State Record, just create a new one.

Choosing a Record Type for State Records

As a general rule, if you want preserve any State Record field values across commits in your program, then you should store those values in a State Record with a Record Type of SQL Table. Only use State Records of the Derived/Work record type to store values that don't need to be accessed across commits. Derived/Work records are, however, an excellent choice for temporary flags and dynamic SQL "containers" that are set and then referenced immediately. Since these values aren't needed later, you don't want to pay the price of saving them to the database at each commit. When you create your State Record in Application Designer, you should have an idea regarding how your State Record will be used. With this information, you can select the appropriate Record Type to build.



For more information regarding record types and the SQL Build process, see Application Designer.

In the context of Application Engine programs, State Records that are Derived/Work records work the same as those which are of a SQL Table type. However, there is one notable distinction: unless you have disabled Restart, Derived/Work records have their field values re-initialized after each commit. Therefore, unless you anticipate this behavior, you're likely to encounter problems. One quick way to diagnose such a problem is to examine a trace. Typically, you'll see %BINDs resolved to values prior to a commit, and then after the commit, they'll have no value.

This behavior is necessary to ensure consistency in the event of an abend and restart. During the restart, Application Engine begins, or restarts, at the point of the last successful commit and restores the values of any State Records with corresponding database tables. Derived/Work records aren't associated with a physical database table, and consequently they can't be restored in the event of a restart.

Application Engine Commit

For the new Application Engine programs that you develop, by default, the commit values at the Section and the Step level are turned off. This means that, by default, *no* commits occur during the program run, except for the implicit commit that occurs after the successful completion of the program.

It's up to you to divide your program into logical units of work by setting commit points within your program. Typically, after Application Engine completes a self-contained task, it might be a good time to commit. How often you apply commits affects how your program will perform in the event of a restart. For set processing programs, commit early and often. For row-based processing, commit after every *n* iterations of the main fetch loop that drives the process.

If you have a Step with a Do While, Do Until, or a Do Select Action, you can set the Frequency option, which drives your commit level. This enables you to set a commit at the Step level that occurs after a specified number of iterations of your looping construct. Application Engine programs commit whenever they are instructed to do so, and because of this there's nothing preventing a developer from having both the frequency option enabled as well as having other individual commits inside of a loop.

The only restriction for batch runs occurs when you have restart enabled, and you are inside a Do Select that is of the Select/Fetch type (instead of "Re-select" or "Restartable"). With Select/Fetch, all commits inside the loop are ignored, including the commit frequency if it's set.

"Restartable" is similar to Select/Fetch, except that you are implying to Application Engine that your SQL is structured in such a way that it will filter out rows that have been "processed" and committed. This allows for a successful restart. One technique for accomplishing this is to have a PROCESSED flag that you check in the WHERE clause of the Do Select, and you perform an UPDATE inside the loop (and before the commit) to set the flag to "Y" on each row you fetch.

The commit logic is designed to do a commit regardless of whether any database changes have occurred. The program commits as instructed by the developer, except when the program is restartable *and* at a point where a commit would affect restart integrity—inside a non-restartable Do Select, for example.

When you have a Step set to commit by default, it means that the Step's commit frequency is controlled by the Section's Auto Commit setting. If the Section is set to commit after every step, then the program commits. Otherwise, the program never commits unless the Step is explicitly set to commit afterward.



The Commit After, Later setting at the Step level enables you to override the Section setting if you don't want to commit after a particular Step.

%TruncateTable Considerations

Some database, such as Oracle, issue an implicit commit for a "truncate" command. If there were other pending (uncommitted) database changes, the results would differ if an abend occurred after the %TruncateTable. To ensure consistency and restart integrity, Application Engine checks the following:

- Whether there are pending changes when resolving a %TruncateTable.
- If the program is at a point where a commit isn't allowed.

If either condition is true, Application Engine issues "DELETE FROM" syntax instead.

"No Rows" Considerations

The default for the No Rows setting (on the Action) is Continue. This setting controls how your program responds when a statement returns no rows. If you leave it set to Continue for commit reasons. In the case of %UpdateStats you may want to set No Rows to Skip Step and thus skip the commit. For example, say you have a single INSERT into a table, followed by an %UpdateStats. If the stats were current before the INSERT and the INSERT affects no rows, then the %UpdateStats is unnecessary.

Re-Using Statements

One of the key performance features of Application Engine is the ability to “re-use” SQL statements by dedicating a persistent cursor to that statement. The following topics describe some items to keep in mind if you decide to take advantage of this feature.

ReUse (SQL Action Property)

Unless you select the ReUse property for you SQL Action, %BIND fields are substituted with literal values in the SQL statement. This means that the database has to recompile the statement every time it is executed.

However, selecting ReUse converts any %BIND fields into real bind variables (:1, :2, and so on), allowing Application Engine to compile the statement once, dedicate a cursor, and re-execute it with new data multiple times. This reduction in compile time can bring dramatic improvements to performance.

In addition, some databases have SQL statement caching. This means that every time they receive SQL, they compare it against their CACHE of previously executed statements to see if they have seen it before. If so, they can reuse the old query plan. This only works if the SQL text matches exactly. This is unlikely with literals instead of bind variables.

When using ReUse, keep the following items in mind:

- ReUse is valid only for SQL Actions.
- Only use ReUse if you do not use Bind variables for column names.
- Only use ReUse if you have no %BINDs in the Select list.
- If the SQL is dynamic, as in you are using %Bind to resolve to a value other than a standard Bind value, *and* the contents of the Bind will change each time the statement gets executed, then you can't enable ReUse. In this situation the SQL is different each time (at least from the database perspective) and therefore can't be "reused."
- If you use the NOQUOTES modifier inside %Bind, there is an implied STATIC. For dynamic SQL substitution, the %Bind has a CHAR field and NOQUOTES in order to insert SQL rather than a literal value. If you enable ReUse, this means the value of the CHAR field gets substituted inline, instead of using a Bind marker (as in :1, :2, and so on). The next time that Application Engine Action executes, the SQL that it executes will be the same as before, even if the value of a static bind has changed.
- If you want to re-prepare a ReUsed statement from scratch, because one of the static binds has changed and the SQL has to reflect that, use %ClearCursor.

If you are running DB2 on OS/390 or AS/400, only use ReUse when you are not using %BINDS as operands of the same operator, as shown in the following example:

```
UPDATE PS_PO_WRK1  
  
SET TAX = %BIND (STATE) + %BIND (FED)
```


This causes Error -417. You can modify the SQL so that you can use ReUse successfully. Suppose your program contains the following SQL:

```
UPDATE PS_PO_WRK1

      SET TAX = 0

      WHERE %BIND(TAX_EXEMPT) = %BIND(TAX_STATUS)
```

If you modify it to resemble the following SQL, ReUse will work:

```
UPDATE PS_PO_WRK1

      SET TAX = 0

      WHERE %BIND(TAX_EXEMPT, STATIC) = %BIND(TAX_STATUS)
```

Bulk Insert

By buffering rows to be inserted, some databases can get a considerable performance boost. Application Engine offers this non-standard SQL enhancement on the following databases: Oracle, Microsoft SQLServer and DB2. We call this feature Bulk Insert. For those database platforms that do not support bulk insert, this flag is ignored.

You should *only* consider using this feature when the INSERT SQL is called multiple times in the absence of intervening COMMITs.

Application Engine ignores the Bulk Insert setting in the following situations:

- SQL is not an INSERT.
- SQL is other than an INSERT/VALUES statement that inserts one row at a time. For instance, the following statements would be ignored: INSERT/SELECT, UPDATE, or DELETE.
- SQL does not have a VALUES CLAUSE.
- SQL does not have a field list before the VALUES clause.

In the previous situations, Application Engine still executes the SQL; it just doesn't take advantage of the performance boost associated with Bulk Insert.

If you want to re-prepare or flush a Bulk Insert statement because one of the static binds has changed and the SQL has to reflect that, use %ClearCursor. A flush also occurs automatically before each commit.

Developing for Restart

Application Engine has the built-in ability to save a program state and restart where it stopped processing in the case of an abend. We refer to this feature as Restart.


It's important to understand that simply leaving **Disable Restart** unselected on the Advanced tab on the Program Properties dialog doesn't guarantee that your program is restartable. In fact, that's just the easy part. You need to design your program appropriately for a restart to work successfully.

This section covers the considerations developers will need to address if they want a program to take advantage of Application Engine's restart capabilities.

Enabling Restart

If you want to enable Restart for an Application Engine program complete the following steps. Keep in mind that enabling Restart as per the following directions does not necessarily mean your program is all set to take advantage of this feature. Read the following sections to discover the additional development issues related to Restart that you need to consider.

To enable Restart

1. Open the program for which you wish to enable Restart.
2. Perform one of the following
 - Select File, Object Properties.
 - Press .
3. Select the **Advanced** tab.
4. Make sure that the **Disable Restart** checkbox is *disabled*, or unchecked.

By default, Restart is enabled for any new program. To disable Restart (for a program performing row-based processing that is inherently restartable), just make sure that the **Disable Restart** checkbox is *enabled*, or checked.



Note. You should also check the Application Engine group on the Process Scheduler tab in the Configuration Manager to view the Disable Restart setting. This setting needs to be off. If either locations specify to Disable Restart (have it selected), then Restart is disabled. You need to have Restart enabled in both locations to enable Restart.

Disabling Restart

There are three ways that you can Disable Restart. They are:

- Checkbox on the Program Properties dialog.
- Checkbox on the Configuration Manager.
- Command line flag to the Application Engine executable (PSAE.EXE).

If in *any* of these three places you've disabled restart, then restart is disabled.

We intend that you would typically leave the Disable Restart checkbox deselected for the Program Properties. Let's say you wanted to test a program you were developing (or multiple programs) and you don't want restart to get in the way. In this case, you could have the Disable Restart checkbox on the Configuration Manager checked, or if you are invoking your program from the command line, you could have the Restart flag enabled.



For more information on the command line parameters used for invoking an Application Engine program, see [Invoking Application Engine Programs](#).

This allows you to test new development and avoid errors about forcing you to restart during your testing. At the same time it prevents you from inadvertently leaving Disable Restart checked in the Program Properties when you moved the program into production.

How Restart Works

By default, Application Engine doesn't perform a COMMIT until an entire program successfully completes. It's up to program developers to set any individual Commits where appropriate. At the Section level, you can choose to Commit after each Step in that Section. And, at the Step level, you can require or defer COMMITs for individual Steps, or you can increase the COMMIT frequency within a Step to *n* iterations of a looping action within a Step, such as a Do Select or Do While.

The Commit level you select, plays a major role in the Restart functionality of a program. Each time that Application Engine issues a Commit with Restart enabled, it records the current state of the program. The recording of the current state that Application Engine performs is referred to as a *checkpoint*.

So, the program behavior changes depending on whether you have Restart enabled or disabled. If you have Restart enabled, whenever there is a commit, Application Engine also performs a checkpoint beforehand. When you have Restart disabled Application Engine only performs a commit.



For more information on Commit see [Application Engine Commit](#).

So, if a failure occurs at any point in the process, the end user can restart the program and expect the program to behave in the following manner:

- Ignore the Steps that have already completed up to the last successful commit.
- Begin processing at the next Step after the last successful commit.

The ability for Application Engine to “remember” what Steps have already completed and which Steps have not, is attributed to an important record called AERUNCONTROL—keyed by Process Instance.

When a program runs, each time Application Engine issues a COMMIT it also saves all of the information required for a program restart in the AERUNCONTROL record. Of course, when an end user restarts the program, the Application Engine checks the AERUNCONTROL values for a particular Process Instance to find what Steps to ignore and where to begin processing.

In addition to offering the obvious advantage of allowing restarts, Application Engine’s Restart feature also affects the design and performance of an Application Engine program. Ultimately, having restart functionality allows the developer to perform COMMITs more often. Well, a developer can always issue a commit as needed. However, having the built in Restart capability gives you more confidence to issue commits freely and flexibly. And, without Restart, you need to be concerned with committing complete chunks of work so that you don’t leave data in a transient state that can’t be easily “undone” if the program abends. With Restart, you can commit as often as you like without worrying about the data.

This reduces the overall impact on other users and processes while the background program is running. This reduced “footprint” can also allow multiple instances of the program to run concurrently (parallel processing), which may be useful for high volume solutions. By “footprint” we are referring to the amount of rows that are locked by the program.

Deciding to Restart

Most of the time, you will want to develop programs to take advantage of the Application Engine restart capabilities. Programs that are good candidates for Restart are those that do a lot of preparation work up front, like joining tables and loading data into temporary work tables. Also, programs that might put your data in an unstable state if they abend during a run should be considered to take advantage of Restart. As a general rule, Restart is essential for programs that primarily do set-based processing.

However, there are special programs for which you may want to *disable* Restart. For example, suppose one the following traits are true for a particular program:

- It’s doing row-by-row processing.
- You don’t want the overhead involved with Application Engine performing a checkpoint during the program run.
- You’re committing after N iterations of a looping construct within a Step, and the SELECT statement driving the loop is composed in such a way that if the program abended and then started again, it would ignore transactions that were already processed in the previous program run. In this sense, the program handles the restart internally in that Application Engine treats

each start of a program as a “fresh” start instead of restarting a previous instance.

If any of these previous characteristics are true for a program, you may want to disable restart. On the other hand, if you decide to create a program that can be restarted, you need to design for a restart at every level, as in the program, Section, and Step level.

When developing for restart, we recommend that you consider the consequences if a program fails and you can’t restart the program. It goes with saying that program failures are unpredictable and can occur at any point in the program. Given the commit structure that you’ve defined for your Application Engine program, would your data get left in a strange state if a failure were to occur after any of the commits? Would it be easy to recover from such a case?

The following topics help you to code a restart strategy into your Application Engine programs.

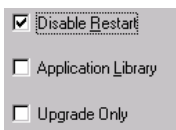
At the Program Level

Application Engine automatically performs all State Record UPDATES. When an Application Engine program starts, it inserts a row in the State Record for the assigned Process Instance. It is then updated by Application Engine whenever it performs a COMMIT to store changed values into the database. The Application Engine State Record row is deleted upon successful completion of the application.

However, if the Application Engine State Record the program uses is a work record, no database UPDATES can be made to the record. Consequently, if you restart the Application Engine program you might get unexpected results because the “memory” was lost when the program abended. In fact, AE re-initializes any state records that are work records at each commit, to ensure consistent behavior during a normal run and a restarted run. Therefore, you may need to make at least one of your State Records a SQL table to contain values that must be retained across commits or in case of an abend.

Of course, the other consideration for programming for restart at the program level is to check *both* the AE Program Properties dialog and the Configuration Manager to make sure that **Disable Restart** is not selected. If Disable Restart is not selected in either of these locations, Application Engine will employ the Restart functionality.

Go to the Advanced tab in the Program Properties dialog, and view the Disable Restart setting, as shown.



Disable Restart in Program Properties

And, also select the Process Scheduler tab in the Configuration Manager to view the Restart setting there.



Disable Restart in Configuration Manger

At the Section Level

One of the properties at the Section level associated with Restart is Section Type. The options for Section Type are Prepare Only and Critical Updates.

If a Section is only preparing data, as in selecting it, populating temporary tables, or updating temporary tables, then the Section Type should be Prepare Only. However, if the Section is actually updating *permanent* application tables in the database, you should select Critical Updates. For instance, with Critical Updates you're typically in a situation where tables are locked, and if the Section does not run successfully to completion, your data could be desynchronized or corrupt.

During runtime, when Application Engine arrives at the first Section specified as Critical Updates, it sets the AE_CRITICAL_PHASE value in the AERUNCONTROL record to *Y*. Once set, the value of AE_CRITICAL_PHASE will remain *Y* until the program completes successfully. When the program completes, the corresponding row in AERUNCONTROL is deleted. For instance, a Prepare Only Section following the Critical Updates Section won't reset the AE_CRITICAL_PHASE value to *N*.

If your program does abend, the end user can check the AE_CRITICAL_PHASE value. If it's *Y*, the operator will know that the Section that failed is critical and that the program should be restarted to ensure data integrity. If AE_CRITICAL_PHASE = *N*, restarting may not be necessary.



Note. As a general rule it's advisable to restart even if AE_CRITICAL_PHASE is set to *N*.

At the Step Level

In your WHERE clause of a Do Select, you need to have some conditions expressed that reduce the answer set returned from the SELECT.

For example,

```
SELECT RECNAME, FIELDNAME
      FROM PS_AERECFIELD
      ORDER BY RECNAME, FIELDNAME
```

If you ran this SELECT as part of a DO Select Action with *Restartable* selected as the Do Select Type, you might process some of the rows twice after a restart. And, if you have specified

Reselect, you would end up with an infinite loop. This is all because there's nothing to reduce the answer set. However, if you modified the SELECT to look more like the following, you could make it *Restartable*.

```
SELECT RECNAME, FIELDNAME

FROM PS_AE_RECFIELD

WHERE RECNAME > %Bind(RECNAME)

OR (RECNAME = %Bind(RECNAME) AND FIELDNAME > %Bind(FIELDNAME))

ORDER BY RECNAME, FIELDNAME
```

A DO Select statement that's coded for *Restartable* can be converted to *Select/Fetch*, but the opposite is not true.

The previous example shows the use of a key column to reduce the answer set. This can be convenient if your record only has one or two key fields. However, if your record has three or four keys, your SQL would become overly complex.

Instead of matching key fields, you could add a switch to the selected table, and then have the processing of the called section modify the switch as it processes the row. In this example, your SELECT could look like the following:

```
SELECT COLUMN1, COLUMN2, . . .

FROM PS_TABLE1

WHERE PROCESSING_SWITCH='N' . . .
```

Abends

Abends, or abnormal ends of your program, can be controlled or uncontrolled.

A controlled abend means that Application Engine exits "gracefully" because of a calculated error condition. Some examples of controlled abends are:

- SQL errors while you have set On Error = Abort.
- PeopleCode return value of if On Return = Abort.
- SQL statement affects no rows and you have set On No Rows = Abort.

In these situations (when Application Engine is in control) the Run Status in Process Monitor reads "Error".

An "uncontrolled" abend occurs in situations where there is a memory violation or a user kills a process. In these cases, the Run Status in Process Monitor shows "Processing".

Using Temporary Tables

It is quite common to develop batch programs that run in a parallel fashion, which means that multiple instances of the same program run simultaneously. This approach is generally referred to as parallel or concurrent processing.

Typically, an organization implements parallel processing when considerable amounts of data need to be updated or processed within a limited amount of time, or "batch window." In most cases, parallel processing is more efficient in environments containing multiple CPU's and partitioned data. Parallel processing also occurs when multiple users submit ad-hoc requests to run the same batch process on transaction data they have just entered.

When batch programs begin running simultaneously, they introduce a significant risk of data contention and deadlocks on common tables and/or temporary tables. Keep in mind that many delivered, PeopleSoft batch applications employ the technique of set processing, and set processing (in most cases) involves extensive use of temporary tables.



For more information on set processing see Set Processing .

For Application Engine programs, PeopleTools provides a feature that enables you to drastically reduce the risk of table contention by dedicating specific instances of temporary tables for each program run. With this feature, you have a pool of temporary table instances, and within that pool some tables are *dedicated* to particular programs, and some instances are *undedicated*, meaning they are "shared." The following sections provide information to help you control how parallel program runs use temporary tables.

Overview

Probably the best way to introduce the concept of managing temporary tables in an effort to run parallel processes is through a hypothetical example. This section provides a high-level illustration of why and how you would go about associating temporary tables with specific Application Engine programs.

Suppose you have an Application Engine program called MYAPPL and it runs against a temporary table named PS_MYAPPLTMP. When you need a SQL statement to SELECT from PS_MYAPPLTMP you would use the PeopleSoft meta SQL construct %Table. For instance, it would appear as

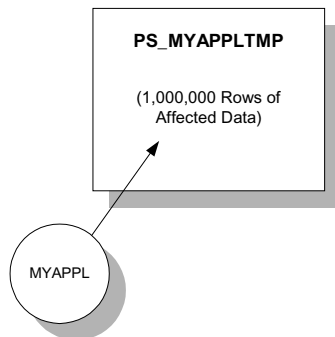
```
%Table (MYAPPLTEMP)
```



For more information on the %Table meta-SQL construct, see Application Engine Meta-SQL.

Let's say that after writing the program, you discover an interesting fact. Due to the volume of data that needs to be updated by your program, there are some situations where

PS_MYAPPLTMP is likely to contain millions of rows. You could ignore the number of rows and choose to run a single instance of your MYAPPL program against a single instance of PS_MYAPPLTMP. But you should consider that although an Application Engine program is suited to run against a table containing so many rows, the performance is not likely to be optimal. In fact, this would be the case with any batch program, such as a COBOL or SQR program.



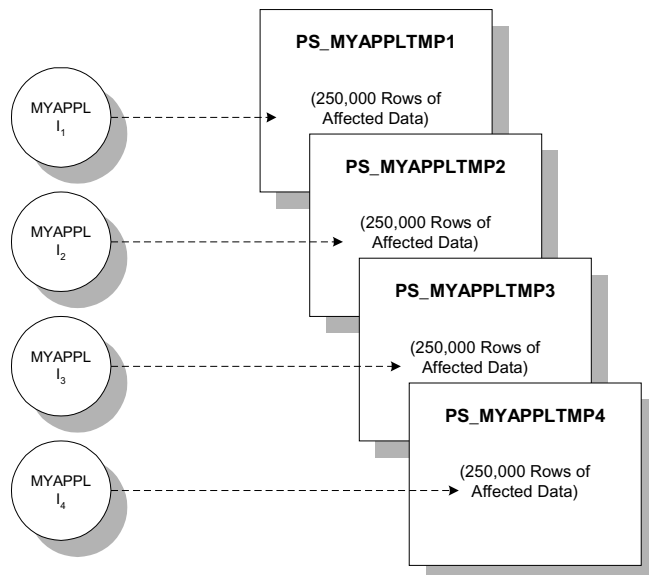
Single Instance of MYAPPL Running against PS_MYAPPLTMP

However, assume that using this approach the program run exceeds four hours, and three hours is the maximum time allotted as the batch window. You are going to need to trim some time off of that program run. You can always tune your SQL or review your set processing algorithms (if you are using any), but you should also explore the possibility of parallel processing.

Parallel processing can enable you to reduce significantly the time required to update the data that currently gets processed while residing in one instance of PS_MYAPPLTMP. Rather than processing all the affected rows of data while they populate one instance of PS_MYAPPLTMP you can partition the input data between multiple concurrent runs of MYAPPL, each with its own dedicated version of PS_MYAPPLTMP. In this example, we'll use four instances of the temporary table.

If you were designing a payroll batch process, you could divide the employee data by last name. For example, employees with last names beginning with A through G get inserted into temporary table one, employees with last names beginning with H-M get inserted into temporary table two, and so on.

The Application Engine runtime program invokes logic to pick one of the available instances. Once each program instance gets matched with an available temporary table instance, the %Table meta-SQL construct resolves to use the corresponding temporary table instance. Run control parameters passed to each instance of the MYAPPL program enable it to identify which input rows "belong" to it, and each program instance inserts the rows from the source table into its assigned temp table instance using %Table.



Multiple Program Instances Running against Multiple Temporary Table Instances

There is no simple switch or checkbox that enables you to turn parallel processing on or off. To implement parallel processing, you need to complete a set of tasks in the order that they appear in the following list. With each task you need to consider important details regarding your specific implementation.

1. **Application Designer.** Define and save all of the temporary table records. You don't need to run the SQL Build process at this point.
2. **Application Engine Designer.** Assign temp tables to Application Engine programs, and set the Instance Counts dedicated for each program. Assign each Temporary table to the Application Engine program(s) using it, set the appropriate number of Instance Counts, and employ the %Table meta-SQL construct so that Application Engine can resolve table references to the assigned temporary table instance dynamically at runtime.
3. **PeopleTools Options.** Set the global instance counts for online programs in PeopleTools Options. Set the number of temporary table instances on the PeopleTools Options page.
4. **Application Designer.** Build all the temporary table records in Application Designer (SQL Build).

The following sections describe the details associated with each task.

Application Designer: Creating Temporary Table Instances

You use Application Designer initially to define your temporary tables, and then you build the tables after you have completed the appropriate tasks in PeopleTools Options and the Application Engine Designer.



For more information on defining records and building SQL tables see Application Designer.

Defining Temporary Tables

PeopleSoft recommends that you insert the `PROCESS_INSTANCE` field as a key into the temporary tables you intend to use with Application Engine. Application Engine expects temporary table records to contain the `PROCESS_INSTANCE` field.

When all instances of a temporary table are in use and the Temp Table runtime options are set to "Continue," PeopleTools will insert rows into the base table using `PROCESS_INSTANCE` as a key. If you opt *not* to include `PROCESS_INSTANCE` as a key field in a temporary table, you should change the Temp Table runtime options to "Abort" in the appropriate Application Engine programs.



Note. In order to take advantage of multiple instances of a temporary table, you must specify that the record definition's Record Type is set to Temporary Table.

Do not use temporary when declaring views for the following reasons:

- If the view is referenced by an Application Engine program that uses dedicated temp tables, the system can't determine which iteration to use.
- The SQL build process needs views to build temporary tables.

Building Temporary Tables

Application Designer builds the temporary table instances at the same time it builds the base table for the record definition. When Application Designer builds a table (as in, Build, Current Object) and the Record Type is Temporary Table, it determines the total number of instances of the temporary table to based on sum of the following two items.

- The Temporary Table Instance (Total) specified on the PeopleTools Options page.
- The total number of instances of that table dedicated within your Application Engine programs.

Application Designer only creates a maximum of 99 temporary table instances, even if the sum exceeds 99 for a particular temporary table.

The naming convention for the temporary table instances is as follows.

```
<base table name>nn
```

Where *nn* is a number between 1 and 99, as in `PS_TEST_TMP23`.



Note. You may elect to take advantage of RDBMS-specific features such as table-spaces and segmentation. For instance, you may want to use the Build process to generate a DDL script, then fine-tune the script prior to execution. For example, you could place different sets of temporary tables on different table-spaces according to instance number.

Application Engine: Managing Temporary Table Instances

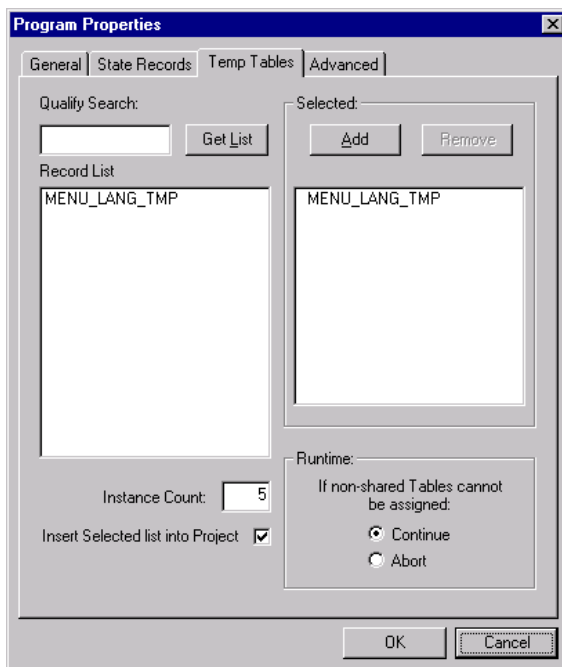
Regardless of the value in the Instance Counts on either the Program Properties dialog or the PeopleTools Options page, you will want to make sure that you have the appropriate records assigned to the appropriate programs. You'll also need to ensure that the SQL inside your Application Engine program contains the correct usage of the %Table construct.



You need to decide the Instance Count value on the Temp Tables tab prior to building the in Application Designer.

Assigning Temporary Tables to Programs

You manage the number of temporary tables assigned to a program by using the Temp Tables tab in the Program Properties dialog.



Temp Tables tab



For more information on the Temp Tables tab, see Application Engine Interface.

This Temp Tables tab enables developers to manage the number of dedicated temporary tables required per program definition. It is the program developer's responsibility to ensure that all the necessary temporary table records are included.

The Instance Count edit box enables you to specify "I want N copies of temporary tables A and B defined for program C." When you change the Instance Count value and click OK, a dialog box appears to remind you that the records must be rebuilt. Anytime you change the instance counts, you need to rebuild the temporary tables to ensure that the right number of instances get created and are available for your programs.



Important! The concept of dedicated temporary tables is isolated to the Application Engine program run. The locking, truncate/delete from, and unlocking are designed to occur within the bounds of an Application Engine program run. Therefore, the system does not keep a temporary table instance available after the Application Engine program run completes.

Adjusting Meta-SQL

A critical step in implementing parallel processing is to make sure that you've included all of the appropriate meta-SQL within the code that your Application Engine program executes. This section contains the specific meta-SQL constructs that you should address.

Referencing Temporary Tables (%Table)

To reference a dedicated temp table, you need to use

```
%Table (record)
```

You can reference any table with %Table, but only those records defined as Temporary Tables get replaced with a dedicated instance table by Application Engine. When you are developing programs that take advantage of %Table, keep the following items in mind:

- All temporary tables should be keyed by Process Instance as a general rule. Also, if you have opted to use the "Continue" option when dedicated tables can't be assigned, Process Instance is *required* as a key field.
- Your choice of indexes on temporary tables is an important consideration. Depending on the use of the temporary table in your program as well as your data profile, the system index(es) may be sufficient. On the other hand, a custom index may be needed instead, or perhaps no indexes are necessary at all. It is worth considering these issues when designing your application. You want to define indexes and SQL that will perform well in most situations, but individual programs or environments may require additional performance tuning during implementation.



Note. The default table name refers to PS_recname, where PS_recname1,2,... refer to the dedicated temporary tables.

As Application Engine resolves any %Table, it checks an internal array to see if a temporary table instance has already been chosen for the current record. If so, then Application Engine substitutes the chosen table name. If not, as in when a record does not appear in the temp table list for the program, then Application Engine uses the base table instance (PS_recname) by default. Regardless of whether %Table is in PeopleCode SQL or in an Application Engine SQL Action the program uses the same physical SQL table.

If you are using dedicated tables, and have elected to continue if dedicated Tables can't be assigned, then all of your SQL references to dedicated temporary tables must include PROCESS_INSTANCE in the WHERE clause.

For synchronous calls to Application Engine, an available instance number will be selected at random according to internal rules. By synchronous, we refer to using the CallAppEngine PeopleCode function; all other methods that you use to invoke Application Engine programs are asynchronous, which means the panel is not “frozen” while the program runs to completion.

Clearing Temporary Tables (%TruncateTable)

You do not need to delete data from a temporary table manually. The temporary tables are truncated automatically when they are assigned to your program. If the shared base table has been allocated because no dedicated instances were available, then Application Engine performs a delete by process instance instead of performing a truncate. In such a case, the PROCESS_INSTANCE is required as a high-level key.

You can perform additional deletes of temporary table results during the run, but you will need to include your own SQL Action that does a %TruncateTable. If the shared base table has been allocated because no dedicated instances were available, then %TruncateTable will be replaced with a delete by process instance instead of a truncate.



Note. You should always use %TruncateTable to perform a mass delete on dedicated temporary tables, especially if the Continue option is in effect.

Even if you have elected to abort if a dedicated table cannot be allocated, you may still use %TruncateTable meta-SQL with dedicated temporary tables. %TruncateTable resolves to either a TRUNCATE or a DELETE by process instance, as needed.

Keep in mind that, the argument of %TruncateTable is a table name instead of a record name. As a result, you'll need to code your SQL as shown in the following example.

```
%TruncateTable(%Table(<recname literal or bind>))
```



Note. You should avoid hard-coded table names inside %TruncateTable since they preclude the possibility of concurrent processing.

Making External Calls

When you call one Application Engine program from another, the assignment of dedicated tables for the called, or "child," program, only occurs if the calling, or parent, program is in a state where a commit can occur immediately.

PeopleTools enables you to commit immediately so that Application Engine can commit the update it performs to lock the temporary table instance. Otherwise, no other parallel process could perform any assignments. In general, this means that you should issue a commit just prior to the Call Section Action.

While making external program calls, keep the following items in mind:

- If the situation is suitable for a commit then the temporary table assignment and the appropriate truncates occur.
- If the situation is not suitable for a commit *and* the called program is set to *continue* if dedicated tables cannot be allocated, then the base tables will be used instead, and a delete by process instance will be performed.
- If the situation is not suitable for a commit *and* the called program is set to *abort* if dedicated tables cannot be allocated, then program execution terminates. This would actually reflect an implementation flaw that you would need to correct.
- If the called Application Engine program shares temporary tables with the calling program, this is allowed. Common temporary tables are the way you share data between the calling and called programs. Application Engine only locks instances of temporary tables that have not already been used during the current program run. Temporary tables that already have an assigned instance will continue to use that same instance.

Also when you make calls to external programs, there are considerations for batch and online program runs to keep in mind.

- **Batch.** For batch runs, in the Program Properties of the root program list all of the temporary tables that any called programs or sections use. This makes sure that the tables get locked sooner and as a single unit. This approach can improve performance, and it ensures that all the tables required by the program are ready before execution starts.
- **Online.** If the online program run is designed to use any temporary tables at *any* point during the CallAppEngine unit of work, then the root program must have at least 1 temp table specified in the Program Properties dialog box. This is true even if the root program doesn't use any temporary tables. This is required so that the system locks the instance number early on to avoid an instance assignment failure after the process has already started processing.

All temporary tables used by a specific program, library, or external section must be specified in that program to ensure that the system issues Truncates (deletes) for the tables being utilized.

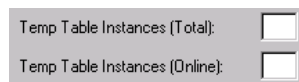
If no temporary tables appear in the root program properties, and Application Engine encounters a %Table reference for a temporary table record, the following error appears:

```
Online AE Process - Invalid attempt to process Temporary Table <record name>
```

PeopleTools Options: Specifying the Number of Temporary Tables

Setting the appropriate number is important for establishing a pool of instances to be used by Application Engine processes invoked “online” from PeopleCode with the CallAppEngine function. In general, the number you choose should be relatively small, as in less than 10. The disadvantage of a larger value is that the extra instances consume space in the database, and can therefore hamper overall performance.

The two parameters on PeopleTools Options related to parallel processing are those having to do with Temp Table Instances. Namely, they are **Temp Table Instances (Total)** and **Temp Table Instances (Online)**.



The image shows a small window titled 'PeopleTools Options'. It contains two input fields. The first is labeled 'Temp Table Instances (Total):' and the second is labeled 'Temp Table Instances (Online):'. Both fields are currently empty.

PeopleTools Options

You need to set the total number of Temp Table Instances, and then you also need to set the number of those instances devoted to your online program runs, such as those invoked by the CallAppEngine PeopleCode function.



In general, the total and online instance counts should be equal unless your application documentation provides specific instructions on setting these values differently.

The following list outlines the reasons why you would set **Temp Table Instances (Online)** to a particular value:

- Application Engine uses this value to identify a range of temporary tables devoted to programs called by CallAppEngine. A randomizing algorithm balances the load for the online process that get assigned to a temporary table devoted to online program execution.
- This value provides backward compatibility for developers who took advantage of the %Table(<record_name>, <instance_number>) approach for *manually* managing temporary table locking.

Temp Table Instances (Total)

The value that you specify in the **Temp Table Instances (Total)** edit box, controls the total number of physical temporary table instances that Application Designer creates for a temporary table record definition when you perform the Build process.

This value indicates the total number of *undedicated* temporary table instances. In the previous example, Application Designer would build 10 physical instances of each record definition of a temporary table when you run the SQL Build process. For example, if you have Temp Table Instances (Total) set to 10, you get 10 tables when you build MYAPPLTEMP in Application Designer: PS_MYAPPLTMP1 through PS_MYAPPLTMP10.



Note. The maximum number of temporary table instances that you can specify is 99. The default value upon installation is 0.

The actual amount of temporary table instances built for a particular temporary table record during the SQL Build process equals the Temp Table Instances (Total) value plus the sum of all the Instance Count values specified in the Application Engine Program Properties dialog.

For example, assume that we have defined a APPLTMPA as a record of type "Temporary" in Application Designer. Let's say that Temp Table Instances (Total) is set to 10 and APPLTMPA appears in the Temp Tables tab in the Program Properties for two Application Engine programs. In one program the Instance Count is set to 3, and in the other the Instance Count is set to 2. When you run SQL Build, PeopleTools builds a total of 15 temporary table instances for APPLTMPA.

Online vs. Batch Program Runs

It's worth mentioning that depending on how you invoke a program the manner in which it uses a temporary table is different. The distinction between invocation methods fall into two categories: online and batch. For instance, an online program might be invoked from a page using PeopleCode, and you typically invoke batch programs by scheduling them to run through Process Scheduler.



Online calls are synchronous calls, and batch calls are asynchronous.



For more information on invoking Application Engine programs, see [Invoking Application Engine Programs](#).

Online Processes

Online processes have their own set of dedicated temporary tables, defined globally on PeopleTools Options.

When you invoke a process online, PeopleTools randomly allocates a single temporary table instance number to programX for *all* its dedicated temp table needs. While programX runs, no other program can use that instance number assigned to programX until programX has run to completion. Any other online process that happens to get the same instance value as programX waits for the lock to be released when programX exits.

In more technical terms, online Application Engine processes employ a simple load-balancing algorithm to divide up many concurrent processes across all the available online instance values. Processes that happen to get the same random value run in series on a first come, first served basis. The higher the number of online instances defined on PeopleTools Options, the less likely it will be for two online processes to get the same value.

Batch Processes

For batch processes, PeopleTools allocates the dedicated table instance numbers on a record-by-record basis. PeopleTools begins with the lowest instance number available for each temporary table until all of the temporary table instances are in use.

When a program ends normally or is cancelled by way of Process Monitor, the system automatically releases the assigned instances. When you invoke programs from the command line or a third party scheduler, you use the Manage Abends page to release instances for abended batch processes in the event that such process can't be restarted.

For batch programs, listing them in the root program means they get locked sooner and as a single unit. This helps performance a bit and ensures that you have all the tables you need before program execution starts.

Run Time Behavior

This section contains topics intended to describe the general run time behavior of Application Engine after you have implemented dedicated temporary tables.

Overview

At runtime, Application Engine assigns dedicated tables for all of the temporary tables that appear in the Temp Tables tab. After assigning the dedicated tables, Application Engine updates a locking table, and designates each assigned table as "in use." Then all of the assigned temporary tables get truncated automatically.

For a particular record, if all the instances are currently in use, and the program is set to continue in such situations, then Application Engine either uses the current shared temporary table instance number (if provided) or it uses the base table.

In either case, Application Engine performs a delete by Process Instance instead of a truncate table. Keep in mind that in such situations temporary table records are required to have PROCESS_INSTANCE as a high-level key.

If an assignment cannot be made and the program is set to abort, then execution will terminate.

Table Locking

The physical, dedicated tables are locked at the time the Application Engine program is loaded into memory. This occurs before the program logic begins. When the program ends normally, Application Engine automatically frees the dedicated temp tables.

The following topics provide more detail on the locking methods.

Restart

For restartable programs, the temporary tables remain locked *across* restarts until the program has completed successfully or until the temporary tables are manually released using Process Monitor or the Manage Abends page.

If restart is *disabled* the temporary tables are unassigned automatically in the event of a controlled abend. However, in the event of a "kill" or a "crash," the tables remain locked, and the tables must be freed using Process Monitor or the Manage Abends page.

If you cancel a process using Process Monitor, PeopleTools frees the temporary tables automatically.

When you use the Manage Abends page, you need to click the Temp Tables button corresponding to the correct Process Instance, and then click the Release button on the Temporary Tables dialog.



When you have manually released the temporary tables from their locked state, you lose any option to restart the program run.

Sharing Temporary Table Data

Dedicated temporary tables do not remain locked across Process Instances. If sequential Application Engine programs need to share data by way of temporary tables, a parent Application Engine program should call the programs that share data.

Sample Implementation

The following scenario describes the runtime behavior of Application Engine while taking advantage of temporary tables.

Suppose that you have Program A and Program B. Assume that there are three temporary table definitions: PS_TMPA, PS_TMPB, PS_TMPC. Let's say that the values on the Temporary Tables tab in Program Properties for each program are as follows:

- **Program A.** PS_TMPA and PS_TMPB are specified as the dedicated temporary tables, and the Instance Count is 4.
- **Program B.** PS_TMPB and PS_TMPC are specified as the dedicated temporary tables, and the Instance Count is 3.

After you run SQL Build in Application Designer, the following inventory of temporary tables appears in the database.

<i>PS_TMPA</i>	<i>PS_TMPB</i>	<i>PS_TMPC</i>
PS_TMPA1	PS_TMPB1	PS_TMPC1
PS_TMPA2	PS_TMPB2	PS_TMPC2
PS_TMPA3	PS_TMPB3	PS_TMPC3

<i>PS_TMPA</i>	<i>PS_TMPB</i>	<i>PS_TMPC</i>
PS_TMPA4	PS_TMPB4	
	PS_TMPB5	
	PS_TMPB6	
	PS_TMPB7	

Because the Instance Count for Program A is 4, the system builds four instances of PS_TMPA and PS_TMPB for Program A to use. Because the Instance Count for Program B is 3, the system builds an additional three instances of PS_TMPB and three instances of PS_TMPC for Program B to use.

Notice that because Program A and Program B are "sharing" PS_TMPB there are seven instances. The system derives this total by adding the Instance Count value from all the programs that share a particular temporary table instance. In this case, the four from Program A and the three from Program B combine to require a total of seven instances of PS_TMPB to be built.

Given that this collection of temporary tables exists in your database, let's say that you invoke Program A. At runtime, Application Engine examines the list of temporary tables dedicated to Program A, and assigns the first available instances to Program A. So, assuming that no other programs are running, Application Engine assigns PS_TMPA1 and PS_TMPB1 to Program A.

Now suppose that shortly after you invoked Program A another user invokes Program B. Again, Application Engine examines the list of temporary tables dedicated to Program B and assigns the first available instances. In this scenario, Application Engine assigns PS_TMPB2 and PS_TMPC1 to Program B. Because Program A is already using PS_TMPB1, the system assigns PS_TMPB2 to Program B.



In essence, the system assigns records, such as TMPA, to programs. The base tables, such as PS_TMPA, are also built, by default, in addition to the dedicated temporary instances. If the Program Properties dialog setting for Temp Tables is set to "continue" when no instances are available, the system uses the base table instead of the dedicated instance.

Application Engine Meta-SQL



Note. This section is intended to provide information regarding the Meta-SQL, System Variables, and macros associated with the development of Application Engine programs. Keep in mind that to see all of the PeopleTools Meta-SQL you should also consult the Meta-SQL documentation that resides within the PeopleCode PeopleBook. Consider this information an "Application Engine Supplement" to what already appears in the PeopleCode PeopleBook. You will notice that some Meta-SQL constructs appear in both documents. For such topics that appear in this section, they contain Application Engine-specific information.

For PeopleSoft applications, dynamic SQL is a requirement since PeopleSoft applications need to be flexible enough to adapt to different database platforms as well as the changing meta-data at your site. You need to be able to change the content and/or meaning of fields and records as needed to reflect your current business rules, and you need the application that runs on top of those records and fields to adapt without any extra development effort.

In the past, Application Engine has addressed some basic dynamic structures, like &CLAUSe and &&RECORD, but we've added significantly to our collection of meta-SQL to provide more flexibility. For instance, in this release we've addressed the following:

- For consistency and clarity, we've converted the existing Application Engine "&" constructs to meta-SQL "%" constructs. For example, &BIND is now %Bind, and &CLAUSe is now %SQL. At the same time, we've opened up some constructs such as %SQL for use outside of Application Engine.
- We've retired the &&RECORD construct, which required an intermediate *Adjustments* step whenever the underlying record was changed, before the program could be run again. Instead, we now have %List metaSQL that offers dynamic substitution of a field list at runtime.
- The new meta-SQL constructs to support the definition of flexible SQL that can adapt automatically to table changes made either by PeopleSoft application developers or by our customers.

To give you an idea of how we intend Meta-SQL to be used within Application Engine, let's take a look at one of the fundamental SQL constructs for set processing:

```
INSERT ( ... ) SELECT .... FROM ... WHERE ...
```



For more information on Set Processing, see Set Processing.

You can now automate the generation of this type of statement using PeopleSoft Meta-SQL constructs. To simplify development we generate the INSERT, SELECT, and (at least part of) the WHERE clause, as determined from the data dictionary. For example,

- The INSERT list is easy—all fields on the record.
- The SELECT list is a little more sophisticated. It looks for a field on each of the JOINed tables until it finds a match. If the field is not found on any of the records in the FROM list, then the default value, as specified in the dictionary, will be used.
- In the WHERE clause, we have the ability to join tables automatically based on common field names.

If you are an experienced Application Engine developer (having used it in previous releases), you will want to make note of the following changes to the Application Engine Meta-SQL/Macros. Of course, you will also want to make note of the other information included in this section.

- &MSG no longer exists. Use the Message Action or the MessageBox function in PeopleCode to add messages to the PS_MESSAGE_LOG table.

- **&&RECORD** no longer exists. Use the **%LIST** and the **%INSERTSELECT** constructs instead.
- If a row is not found, **SELECT** buffers can now retain their prior values. **%Select** preserves field values, and **%SelectInit** resets field values. So, you might want to review your code to find areas that might benefit from the use of **%Select** instead of **%SelectInit**.



Note. The SQL Editor doesn't validate all of the meta-SQL constructs, such as **%BIND** and **%SELECT**. Don't be alarmed if the editor claims these constructs are invalid.

Application Engine Meta-SQL

The following topics discuss the meta-SQL constructs that are unique to Application Engine processing.

%Bind

The Application Engine function, **%Bind**, is used to *retrieve* a field value from a State Record. The **%Bind** function can be used anywhere in a SQL statement.

The syntax for **%BIND** is:

```
%BIND ( [recordname.] fieldname [, NOQUOTES] [, NOWRAP] [, STATIC] )
```

When executed, **%BIND** returns the value of the State Record field identified within its parentheses.

Parameters

The parameters are as follows:

- **recordname.** The name of a State Record. If you do not specify a particular State Record, Application Engine uses the default State Record to resolve the **%BIND(fieldname)**.
- **fieldname.** The field defined in the State Record.
- **NOQUOTES.** If the field specified is a character field, its value will automatically be enclosed in quotes unless you use the **NOQUOTES** option. Use **NOQUOTES** to include a dynamic table and field name reference, even an entire SQL statement or clause, in an Application Engine SQL Action.
- **NOWRAP.** If the field is a date, the system automatically wraps its value in **%datein()** or **%dateout()**, unless you use the **NOWRAP** option. The same is true for time and date-time fields. Therefore, If the State Record field is populated correctly, you don't need to worry about the inbound references, although you can suppress the "In" wrapping with the **NOWRAP** modifier inside the **%BIND**. Furthermore, Application Engine skips the "In" wrapper if the **%BIND(date)** is in the select field list of another **%Select** statement. This is because the bind value is already in the "out" format, and the system "selects" it into another State Record field

in memory. As such, in this circumstance there is no need for either an "Out" wrapper or an "In" wrapper. For example,

First SQL Action:

```
%Select (date_end)

SELECT %DateOut (date_end )

FROM PS_GREG
```

Second SQL Action:

```
INSERT INTO ps_greg

VALUES (%Bind (date_end))
```

- **STATIC.** The STATIC parameter allows you to include a “hard-coded” value in a reused statement. For %BINDs that contain dynamic SQL, this parameter must be used in conjunction with NOQUOTES for proper execution of a re-used statement.

For example,

```
UPDATE PS_REQ_HDR

SET IN_PROCESS_FLG = %BIND (MY_AET.IN_PROCESS_FLG) ,

PROCESS_INSTANCE = %BIND (PROCESS_INSTANCE)

WHERE IN_PROCESS_FLG = 'N'

AND BUSINESS_UNIT || REQ_ID

IN (SELECT BUSINESS_UNIT || REQ_ID

FROM PS_PO_REQRCON_WK1

WHERE PROCESS_INSTANCE = %BIND (PROCESS_INSTANCE))
```

In the previous example, %BIND(PROCESS_INSTANCE) assigns the value of the field PROCESS_INSTANCE in the default State Record to the PROCESS_INSTANCE field in table PS_REQ_HDR.

The %BIND function is also used in a WHERE clause to identify rows in the table PS_PO_REQRCON_WK1, in which the value of PROCESS_INSTANCE equals the value of PROCESS_INSTANCE in the default State Record.

When using %BIND, keep the following items in mind:

- Typically, when you use %BIND to provide a value for a field or a WHERE condition, the type of field in the State Record that you reference with %BIND must match the field type of the corresponding database field used in the SQL statement.
- On most platforms, you can't use a literal to populate a LONG VARCHAR field. You should

use the %BIND(record.fieldname) construct.

- In the case of an external call to a section in another program, if the called program has its own default State Record defined, then Application Engine uses that default State Record to resolve the %BIND(fieldname). Otherwise, the called program will inherit the calling programs default State Record.
- All fields referenced by a %SELECT command must be defined in the associated State Record.
- You *must* use the Date, Time and DateTime output wrappers in the SELECT list that populates the State Record fields. This ensures compatibility across all supported database platforms. For example,

First SQL Action:

```
%Select (date_end)

      SELECT %DateOut (date_end )

      FROM PS_EXAMPLE
```

Second SQL Action:

```
INSERT INTO PS_EXAMPLE

      VALUES (%Bind (date_end) )
```

Handling Bind Variables and Date Wraps (PeopleCode SQL vs. Application Engine SQL)

The behavior of bind variables within Application Engine PeopleCode and "normal" PeopleCode is exactly the same.

On the other hand, if you compare Application Engine SQL to PeopleCode (of any type), then the system handles bind variables differently. There are the following options.

Option 1

If you use the following approach

```
AND TL_EMPL_DATA1.EFFDT <= %P(1))
```

Then in PeopleCode you issue

```
%SQL(MY_SQL, %DateIn(:1))
```

This assumes that you have referenced the literal as a bind variable.

Or in Application Engine SQL you issue:

```
%SQL(MY_SQL, %Bind(date_field))
```


Option 2

On the other hand if you elect to use the following approach:

```
AND TL_EMPL_DATA1.EFFDT <= %datein(%P(1))
```

Then in PeopleCode you issue:

```
%SQL(MY_SQL, :1)
```

This assumes that you have referenced the literal as a bind variable.

Or in Application Engine SQL you issue:

```
%SQL(MY_SQL, %Bind(date_field, NOWRAP))
```

%ExecuteEdits

The %ExecuteEdits construct is Application Engine-only meta-SQL. You can't use it in COBOL, SQR, or PeopleCode—not even Application Engine PeopleCode. This function allows Application Engine to support data dictionary edits in batch.

The syntax is:

```
%ExecuteEdits(<type>, recordname [alias][, field1, field2, ...])
```

where <type> consists of any combination of the following (added together):

```
%Edit_Required
```

```
%Edit_YesNo
```

```
%Edit_DateRange
```

```
%Edit_PromptTable
```

```
%Edit_TranslateTable
```

The *recordname* parameter specifies the record used to obtain the data dictionary edits, and the optional list of fields is used to restrict the edits to a subset of the record's fields.

For example, suppose you want to insert rows with missing or invalid values in three specific fields, selecting data from a temporary table but using edits defined on the original application table. Notice the use of an alias or “correlation name” inside the meta-SQL.

```
INSERT INTO PS_JRNL_LINE_ERROR (...)

SELECT ... FROM PS_JRNL_LINE_TMP A

WHERE A.PROCESS_INSTANCE = %BIND(PROCESS_INSTANCE)

      AND %EXECUTEEDITS(%Edit_Required + %Edit_PromptTable, JRNL_LINE A,
BUSINESS_UNIT, JOURNAL_ID, ACCOUNTING_DT)
```

If you need to update rows in a temporary table that have some kind of edit error, you can use custom edits defined on the temporary table record.

```
UPDATE PS_PENDITEM_TAO

SELECT ERROR_FLAG = 'Y'

WHERE PROCESS_INSTANCE = %BIND(PROCESS_INSTANCE)

AND %EXECUTEEDITS(%Edit_Required + %Edit_YesNo + %Edit_DateRange +
%Edit_PromptTable + %Edit_TranslateTable,

PENDINGITEM_TAO)
```

When using %ExecuteEdits, keep the following items in mind.

- Consider performance carefully when using this construct. Prompt table and translate table edits are particularly expensive, since they involve correlated sub-queries. Run a SQL trace at execution time so that you can view the generated SQL. Look for opportunities where it can be optimized.
- In general, %ExecuteEdits is best used on a temporary table. If you need to run this against a "real" application table, you should provide WHERE clause conditions to limit the number of rows to include only those that the program is currently processing. We suggest that you process the rows in the current set all at once rather than processing them row-by-row.
- With %ExecuteEdits you can't use work records in a batch, set-based operation. So, all higher-order key fields used by prompt table edits must exist on the record that your code intends to edit, and the field names must match exactly. For example,

```
%ExecuteEdits(%Edit_PromptTable, MY_DATA_TMP)
```

The record MY_DATA_TMP contains field STATE with prompt table edit against PS_REGION_VW, which has key fields COUNTRY and REGION. The REGION field corresponds to STATE, and COUNTRY is the higher-order key. In order for %ExecuteEdits to work correctly, the MY_DATA_TMP record must contain a field called COUNTRY. It's permissible for the edited field (STATE) to use a different name because Application Engine always references the last key field (ignoring EFFDT).

- Restrict the number *and* type of edits to the minimum required. Don't perform edits on fields that are known to be valid, or that will be defaulted later in the process. Also, consider using a separate record with edits defined specifically for batch, or provide a list of fields to be edited.

%Select

A %SELECT is required at the beginning of any and all SELECT statements. For example, you need one in the Flow Control Actions as well as one in the SQL Actions that contain a SELECT. The %SELECT function identifies the State Record fields to hold the values returned by the corresponding SELECT statement. In other words, you use %SELECT to pass values to the State Record buffers.

You use the %SELECT construct to pass variables to the State Record, and you use the %BIND construct to retrieve or reference the variables.

The syntax for %SELECT is:

```
%SELECT(statefield1[, statefield2]...[, statefieldN])

SELECT field1[, field2]...[, fieldN]
```

The *statefields* must be valid fields on the State Record (may be fieldname or recordname.fieldname, as with %BIND) and *fields* must be either valid fields in the FROM table(s) or hard-coded values.

Consider the following sample statement:

```
%SELECT (BUSINESS_UNIT, CUST_ID)

SELECT BUSINESS_UNIT, CUST_ID

FROM PS_CUST_DATA

WHERE PROCESS_INSTANCE = %BIND (PROCESS_INSTANCE)
```

The following steps illustrate the execution of the previous statement:

1. **Resolve Bind Variables.** The string %BIND(PROCESS_INSTANCE) is replaced with the value of the State Record field called PROCESS_INSTANCE.
2. **Select.** Execute the following SQL SELECT statement.
3. **Fetch.** A SQL FETCH is performed. If a row is returned, the State Record fields BUSINESS_UNIT and CUST_ID are updated with the results. If the Fetch does not return any rows, all fields in the %SELECT retain their prior values.



Note. All fields referenced by a %SELECT command must be defined in the associated State Record. Also, aggregate functions always return a row so they will always cause the State Record to be updated. As such, for aggregate functions there is no difference whether you use %SelectInit or %Select.

%SelectInit

This meta-SQL construct, %SelectInit, is identical to %SELECT barring the following exception. If the SELECT returns no rows, %SelectInit reinitializes the buffers. In the case of a %SELECT and no rows are returned, the State Record fields retain their previous values.



Note. PeopleSoft added this construct primarily for backwards compatibility to the &SELECT construct of PeopleTools 7.5. The conversion program substitutes %SelectInit for &SELECT so that after conversion the behavior does not change. However, for performance reasons, PeopleSoft recommends using %SELECT if the buffer initialization is not necessary.



For more information on the conversion program and manually converting %SelectInits to %SELECTs, see your upgrade documentation where upgrading/converting previous Application Engine programs is discussed.



Note. Aggregate functions always return a row so they will always cause the State Record to be updated. As such, for aggregate functions there is no difference whether you use %SelectInit or %Select.

%SQL

When you use %SQL in a statement, Application Engine replaces it with the specified SQL object. This allows commonly used SQL text to be shared among Application Engine and PeopleCode programs alike. In Application Engine, you use %BIND to specify your bind variables. In PeopleCode SQL, you can use

```
:record.field
```

or

```
:1
```

If you create SQL objects that you plan to share between Application Engine and PeopleCode programs, the %SQL construct allows you to pass parameters for resolving bind variables without being concerned with the difference in the bind syntax that exists between Application Engine and PeopleCode. However, keep in mind that the "base" SQL statement that uses %SQL to represent a shared object with binds needs to be tailored to Application Engine or to PeopleCode.

For example, let's assume that your SQL is similar to the following:

```
UPDATE PS_TEMP_TBL SET ACTIVE = %BIND(MY_AET.ACTIVE)

WHERE PROCESS_INSTANCE = %ProcessInstance
```

That would not be valid if the SQL executed in PeopleCode. However, if you define your SQL as shown:

```
UPDATE PS_TEMP_TBL SET ACTIVE = %P(1)
```

```
WHERE PROCESS_INSTANCE = %ProcessInstance
```

You could use parameters in %SQL to insert the appropriate bind variable.

From Application Engine, the "base SQL" or source statement might look like the following:

```
%SQL (SQL_ID, %BIND (MY_AET.ACTIVE))
```

The PeopleCode SQL may appear as the following:

```
%SQL (SQL_ID, :MY_AET.ACTIVE)
```



Note. You only can use %SQL to reference SQL Objects created directly in Application Designer. For instance, you can not use %SQL to reference SQL that resides within a Section in an Application Library. Common SQL should be stored as a proper SQL Object.



For more information on the syntax for %SQL, see %SQL in the PeopleCode Meta-SQL, PeopleCode Developer's Guide.

%Table

The %Table function returns the SQL table name for the record specified with *recname*. The basic syntax is as follows:

```
%Table (recname)
```

For example,

```
%Table (ABSENCE_HIST)
```

Returns the record PS_ABSENCE_HIST.

If the record is a temporary table and the current process has a temporary table instance number specified, then %Table resolves to that instance of the temporary table PS_ABSENCE_HIST nn , where nn is the instance number).

This function can be used to specify temporary tables for running parallel Application Engine processes.

You can use the %Table function when you want to be able to run the same Application Engine program, in parallel, across different subsets of the data.



For more information on managing temporary table images and parallel processing for Application Engine programs see Using Temporary Tables.

%TruncateTable

The %TruncateTable meta-SQL construct is functionally identical to a DELETE SQL statement with no WHERE clause, but it is faster and requires less resources on databases that support bulk deletes. If you're familiar with COBOL this construct is an enhanced version of the COBOL meta-SQL construct with the same name.

Some database vendors have implemented bulk delete commands that decrease the time required to delete all the rows in a table by not logging rollback data in the transaction log. For the databases that support these commands, Application Engine replaces %TruncateTable with "TRUNCATE TABLE" SQL. For the other database types, %TruncateTable gets replaced with "DELETE FROM" SQL.

Unlike the COBOL version, Application Engine determines if a commit is possible prior to making the substitution. If a commit is possible, Application Engine makes the substitution and then forces a checkpoint and commit after the successful execution of the delete.

If a commit is not possible, Application Engine replaces the meta-SQL with a DELETE FROM string. This ensures restart integrity when your program runs against a database where there is an implicit commit associated with TRUNCATE TABLE or where rollback data is not logged.

The basic syntax for %TruncateTable is

```
%TruncateTable(table name)
```

For example,

```
%TruncateTable(PS_PO_WRK1)
```

For databases that either execute an implicit COMMIT for %TruncateTable or require a COMMIT before and/or after this meta-SQL, replaces %TruncateTable with an unconditional DELETE in the following circumstances:

- A Commit is not allowed, as in within an Application Engine program called from PeopleCode.
- The program issues a Non-Select SQL statement since the last COMMIT occurred. In such a situation, data is likely to have changed.
- You are deferring COMMITs in a SELECT/FETCH loop within a restartable program.

%UpdateStats

Application Engine replaces this meta-SQL construct with a platform dependent SQL statement that updates the system catalog tables used by the database optimizer in choosing optimal query plans. We intend that you use this construct after your program has inserted large amounts of data into a temporary table that will be deleted before the end of the program run. This saves you from having to use "dummy" seed data for the temporary table and having to update statistics manually.

Using %UpdateStats

The basic syntax for %UpdateStats is

```
%UpdateStats (record name)
```

For example,

```
%UpdateStats (PO_WRK1)
```

When you call %UpdateStats from an Application Engine program, keep the following in mind:

For databases that either execute an implicit COMMIT for %UpdateStats or require a COMMIT before and/or after this meta-SQL, Application Engine skips %UpdateStats in the following circumstances:

- A Commit is not allowed, as in within an Application Engine program called from PeopleCode.
- The program issues a Non-Select SQL statement since the last COMMIT occurred. In such a situation, data is likely to have changed.
- You are deferring COMMITs in a SELECT/FETCH loop in a restartable program. Application Engine skips %UpdateStats even if the previous condition is false.

The following table shows how the %UpdateStats construct gets resolved by the supported database systems.

<i>RDBMS</i>	<i>Resolution</i>
Oracle	ANALYZE TABLE <name> ESTIMATE STATISTICS
Informix	UPDATE STATISTICS MEDIUM FOR TABLE <name>
Microsoft SQL Server	UPDATE STATISTICS <name>
Sybase	UPDATE STATISTICS <name>
DB2 for UNIX	RUN STATISTICS FOR TABLE (<name>)
DB2 for OS/390	N/A

Using %UpdateStats with COBOL

You can issue the %UpdateStats construct from SQL embedded in your COBOL programs as well. You may want to consider employing this construct in any COBOL modules that you call from Application Engine.

When issuing %UpdateStats from COBOL, the syntax is slightly different.

```
%UpdateStats (<tablename>)
```

When you issue this construct from PeopleTools, the parameter is <record name>. Make note of this distinction.

%UpdateStats Considerations

The following table alerts you to some potential issues that you may encounter when using %UpdateStats.

<i>RDBMS</i>	<i>Consideration</i>
Microsoft SQL Server Sybase UDB	<p>PeopleSoft forces a Commit before and after the %UpdateStats statement.</p> <p>Therefore, we skip (null op) this meta-SQL if a Commit is not allowed. For instance, a Commit is not allowed in the following situations:</p> <p>The Application Engine program is running online, as in not in batch mode.</p> <p>You have issued non-Select/Fetch SQL (in which the data is likely to change) since the last Commit.</p> <p>You are deferring Commits in a SELECT/FETCH loop within a restartable program.</p>
Oracle	<p>Oracle has an implicit Commit after the %UpdateStats statement executes.</p> <p>Therefore, we skip (null op) this meta-SQL if a Commit is not allowed. For instance, a Commit is not allowed in the following situations:</p> <p>The Application Engine program is running online, as in not in batch mode.</p> <p>You have issued non-Select/Fetch SQL (in which the data is likely to change) since the last Commit.</p> <p>You are deferring Commits in a SELECT/FETCH loop within a restartable program.</p>
DB2 for OS/390	<p>For %UpdateStats to work correctly on DB2 for OS/390 you'll need to complete a considerable amount of DBA work. For this reason we recommend disabling %UpdateStats for OS/390.</p>
Informix IBM UDB	<p>%UpdateStats will lock the table being analyzed on UDB and Informix (any other platforms?). Therefore, we suggest that this meta-SQL only be used on tables that are not like to be concurrently accessed by other applications and users. A perfect use thereof would be to analyze AE dedicated temp tables.</p>

RDBMS	Consideration
All	%UpdateStats will consume an enormous amount of time and db resources if run against very large tables. Therefore, we suggest that permanent data tables be analyzed outside of application programs. Also, if temp tables are likely to grow very large during a batch run, we recommend only running the batch program with %UpdateStats enabled to seed the statistics data or when the data composition changes dramatically

Disabling %UpdateStats

If you want to disable the %UpdateStats functionality, you can do so in the following ways:

- Include the following parameter on the command line when running an Application Engine program:


```
-DBFLAGS 1
```
- Click the Disable DB Stats check box in the Shared Flags group on the Process Scheduler tab in the Configuration Manager.
- Change the Dbflags=0 parameter in the Process Scheduler configuration file (or PSADMIN) to Dbflags=1.

Application Engine Macros

The following topics cover Application Engine macros.

%ClearCursor

Use the %CLEARCURSOR function to recompile a re-used statement and reset any STATIC %BINDs. The proper syntax is as follows:

```
%CLEARCURSOR(program, section, step, action)
```

or

```
%CLEARCURSOR (ALL)
```

Using (ALL) clears all cursors.

For the action parameter, the following table contains the valid values.

Value	Action Type
D	Do Select
H	Do When
N	Do Until

Value	Action Type
S	SQL
W	Do While

When you use the %CLEARCURSOR function keep the following items in mind:

- It must be located at the beginning of the statement.
- It can be the only function or command contained in the statement.
- The action parameter must specify an action that executes SQL.

%Execute

The %EXECUTE function allows you to execute RDBMS-specific commands from within your Application Program.

Also, the %EXECUTE construct allows you to include multiple statements in a single Application Engine Action without encountering RDBMS-specific differences. For instance, there are instances where you could code a single Application Engine Action to contain multiple SQL statements and they may run successfully on one database platform. However, if you attempt to run the same code against a different database platform you may encounter errors or skipped SQL.

The syntax for %Execute is as follows:

```
%EXECUTE ( [ / ] )
command1 { ; | / }
command2 { ; | / } ...
commandN { ; | / }
```

By default, Application Engine expects a semi-colon to be used to delimit multiple commands within an %EXECUTE function statement. You can instruct Application Engine to use a forward slash (/) delimiter instead by placing a forward slash inside the function parentheses.

For example, this allows you to use an Oracle PL/SQL block in an %EXECUTE statement, as shown below:

```
%EXECUTE (/)
DECLARE
  counter INTEGER;
BEGIN
  FOR counter := 1 TO 10
    UPDATE pslock SET version = version + 1;
  END FOR;
END;
/
```



When you use the %EXECUTE function, it must be located at the beginning of the statement and can be the only function or command contained in the statement. The Action type must be “SQL”.

%Next and %Previous

This construct is valid in any Application Engine SQL Action, and we intend that you use it when performing sequence-numbering processing. Typically, you’d use it in place of a %Bind. These constructs use the current value of the number field as a Bind variable, and then increment (%Next) or decrement (%Previous) the value *after* the statement is executed successfully. By “number” field, we are referring to the numeric field on the State Record that you have initially set to a particular value (as in ‘1’ to start).

If the statement is a SELECT and no rows are returned, the field value is not changed. The substitution rules are the same as for %Bind. For example, if re-use is enabled, then the field is a true bind (:n substituted). Otherwise, inline substitution occurs.

For example, you could use these constructs in an UPDATE statement within a Do Select.

Step01

- Do Select

```
%SELECT(field1, field2, ...) SELECT key1, key2, ... FROM PS_TABLE WHERE ...
ORDER BY key1, key2, ..."
```

- SQL

```
UPDATE PS_TABLE SET SEQ_NBR = %NEXT(seq_field) WHERE key1 = %BIND(field1) AND
key2 = %BIND(field2) ...
```

With a Do Select the increment/decrement occurs once per execution, not once for every fetch. So unless your Do Select is of the "Reselect" type, the value only gets changed on the *first* iteration of the loop. On the other hand, with Reselect or Do While/Until actions, every iteration re-executes the Select and then fetches one row, and with these types of loops, the value changes on *every* iteration.

%RoundCurrency

This meta-SQL is an enhanced version of the Application Engine &ROUND construct that appeared in previous releases. The %ROUND CURRENCY function rounds an amount field to the currency precision specified by the field’s **Currency Control Field** property—as defined in Application Designer’s **Record Field Properties** dialog. For this function to work, you must have the **Multi-Currency** option selected in the PeopleTools Options panel. As with all Application Engine macros described in the current section, this construct is only valid in Application Engine SQL; it is not valid for SQLExecs or view text.

The syntax for %RoundCurrency is:

```
%RoundCurrency( <EXPRESSION>, [ALIAS.]<CURRENCY_FIELD>)
```

You can use this macro in the SET clause of an UPDATE statement or the SELECT list of an INSERT/SELECT statement. The first parameter is an arbitrary expression of numeric values and/or columns from the “source” table(s) that computes the monetary amount to be rounded. The second parameter is the control currency field from a particular “source” table (the UPDATE table, or a table in the FROM clause of an INSERT/SELECT statement). This field identifies the corresponding currency value for the monetary amount.



Note. Keep in mind that the “As Of Date” of the Application Engine program will be used for obtaining the currency-rounding factor. The currency-rounding factor is determined by the value of DECIMAL_POSITIONS on the corresponding row in PS_CURRENCY_CD_TBL, which is an effective-dated table.

If multi-currency is not in effect, then the result will be rounded to the precision of the amount field (either 13.2 or 15.3 amount formats are possible).

For example,

```
UPDATE PS_PENDING_DST
    SET MONETARY_AMOUNT =
        %RoundCurrency( FOREIGN_AMOUNT * CUR_EXCHNG_RT, CURRENCY_CD)
    WHERE GROUP_BU = %Bind(GROUP_BU) AND GROUP_ID = %Bind(GROUP_ID)
```

Application Engine System (Meta) Variables

In addition to the meta-SQL and macros previously discussed, there are some text-substitution variables—similar to meta-SQL—that are unique to Application Engine.

%AeProgram

Returns a quoted string containing the currently executing Application Engine program name.



Note. Be careful when using the Call Section Action. For example, if Program A calls Program B, and you’re using this macro in Program B, then the value ‘B’ will be returned.

%AeSection

Returns a quoted string containing the currently executing Application Engine Section name.

%AeStep

Returns a quoted string containing the currently executing Application Engine Step name.

%JobInstance

Returns the numeric (unquoted) Process Scheduler Job Instance.

%ProcessInstance

Returns the numeric (unquoted) Process Instance.

%ReturnCode

Returns the numeric (unquoted) return code of the last SQL operation performed.

%RunControl

Returns a quoted string containing the current Run Control identifier. The Run Control ID is available to your program, when using %RunControl, regardless of whether there's a row in the AEREQUEST table.

%AsOfDate

Returns a quoted string containing the “AsOfDate” used for the current process.

%Comma

Returns a comma. This is useful in those cases where you need to use a comma, but commas are not allowed due to the parsing rules. For example, you might use this if you wanted to pass a comma, as a parameter, to the %SQL meta-SQL function.

%LeftParen

Returns a left parenthesis. Usage is similar to %Comma.

%RightParen

Returns a right parenthesis. Usage is similar to %Comma.

%Space

Returns a single space. Usage is similar to %Comma.

%SQLRows

This construct can be used in any Application Engine SQL statement, but the underlying value is only affected by SQL Actions. It is not affected by the program flow control Actions: Do When,

Do Select, Do While, and Do Until. Regardless of where it's used, the semantic remains the same: "How many rows were affected by the last SQL action?"

For SELECT statements, the value can only be 0 or 1: row not found or rows found, respectively. It does not reflect the actual number of rows that meet the WHERE criteria. In order to find the number of rows that meet the WHERE criteria, you need to code a SELECT COUNT(*).

%List



The %List construct is not Application Engine specific, however, there are some restrictions that you should be aware of when using %List in your Application Engine programs.

The %List function expands into a list of field names, delimited by commas. Which fields are included in the expanded list depends on the parameters passed to the function.

PeopleSoft has restricted the use of %List within your Application Engine programs. You can still use the construct; these restrictions merely prevent any improper use of %List. PeopleSoft supports %List in Application Engine SQL Actions with the following restriction.

When using %List in an insert/select or insert/values or %Select statement, you must have matching pairs of %List (or %ListBind) in the target and source field lists, using the same list type argument and record name to ensure consistency.



For more information on %List see PeopleCode Developer's Guide..

Using PeopleCode in Application Engine Programs

Inserting PeopleCode within Application Engine programs provides a lot of opportunity for developers to reuse common function libraries and improve performance. It also can simplify development in some instances, as well. In many cases, a small PeopleCode program used instead of a Do When Action, for example, can replace a more complicated SQL statement.

There are few restrictions in terms of how you take advantage of PeopleCode in your Application Engine program, however, as with most software development tools, there is an "intended" use for a given tool. Application Engine PeopleCode is an excellent way to build dynamic SQL, perform simple IF/ELSE edits, set defaults, and perform other tasks that don't require a trip to the database. There are appropriate and inappropriate uses of the PeopleCode within PeopleCode Actions.

In this section, we introduce you to some of the ways you can include PeopleCode in your Application Engine programs. And, most importantly, we inform you of how not to use PeopleCode in the context of an Application Engine program.



Note. This section is not intended to replace the information that appears in the PeopleCode PeopleBook. Rather, this discussion is an Application Engine extension to the syntax documentation that appears in the PeopleCode PeopleBook. If you are planning to use PeopleCode in your Application Engine programs, we suggest that you become very familiar with the PeopleCode documentation first. While reading through these guidelines, keep in mind that in all cases, you should refer to the PeopleCode PeopleBook for complete syntax, parameters, methods, returns, and so on. Consider the following information guidelines specific to Application Engine.

Application Engine's Purpose

Now that Application Engine supports PeopleCode Actions, the door is open for Application Engine to be used to solve a wider range of business problems than in previous releases. We expect that developers will exploit the use of PeopleCode in all appropriate situations to add new flexibility to Application Engine programs. However, when using PeopleCode in Application Engine programs, it's important not to lose sight of Application Engine's original (and current) purpose.

Application Engine is a development tool that allows you to develop programs that execute SQL, in batch and online mode, against your data in a procedural and dynamic structure. The key word in the previous statement is "SQL." Despite Application Engine's richer functionality, its primary purpose is still to run SQL against your data. In other words, Application Engine is *not* intended to execute programs that include nothing but PeopleCode Actions.

This is mainly due to the fact that Application Engine is most powerful when it executes SQL-based processing against your database. Since PeopleCode is an interpreted language, there is an inherent extra performance overhead when compared to a compiled language. Consequently, a batch program comprised entirely of PeopleCode should be avoided in most cases.

For the most part, PeopleSoft recommends that you use PeopleCode for setting IF, THEN, ELSE logic constructs, performing data "preparation" tasks, and building dynamic portions of SQL, while still relying on SQL to complete the bulk of the actual program processing. We also expect that you will use PeopleCode as a way to re-use online logic that's already developed. And, of course, PeopleCode is the vehicle for leveraging the new technologies, such as Application Messaging, Component Interfaces, and Business Interlinks.

Most programs need to check that a certain condition is TRUE prior to executing a particular Section. For example, if the hourly wage is less than or equal to 'X', do Step A; if not, fetch the next row. And, in certain instances, you'll need to modify variables that exist in a State Record. PeopleCode allows you to set State Record variables dynamically.

Environment Considerations

When writing or referencing PeopleCode in a PeopleCode Action, you need to consider the environment in which the Application Engine program will run. By environment, we are not referring to your RDBMS and workstation configurations but rather the differences between

online and batch modes. It's very important to keep in mind that Application Engine programs usually run in batch mode, and, consequently, your PeopleCode can't access panels or controls as it can while running in online mode. In short, any PeopleCode operations that manipulate panels will not run successfully. Even if you invoke your Application Engine program "online" from a Record or a Panel using the CallAppEngine PeopleCode function, the Application Engine PeopleCode still does not have direct access to the Panel buffers.



Note. Application Engine programs can't access Page buffers.

Any RECORD.FIELD references that appear in a PeopleCode Action can only refer to fields that exist on an Application Engine State Record. Panel buffers, controls, and so on are still inaccessible even if you define the Panel records as State Records on the Program Properties dialog. An Application Engine program can only access State Records or other Objects you create in PeopleCode.

However, PeopleSoft realizes that in many situations it is necessary to pass data from a panel buffer to an Application Engine program. If your program requires this functionality, you have the following options.

Passing Parameters through CallAppEngine

For individual panel fields and simple PeopleCode variables, such as numbers and strings, you can use the CallAppEngine PeopleCode function to pass values as parameters. You can do this by performing the following:

- Declare a Record Object in PeopleCode, as in

```
Local Record &MyRecord;
```

- Assign the Record Objects to any State Record that you want to pass to the Application Engine program. Record Objects are parameters to the CallAppEngine function.
- Set the appropriate values on that State Record.
- Include the Record Object in the function call.

After these values get set in the State Record, all the Actions in a particular program can use the values, not just the PeopleCode Actions.



For more information on CallAppEngine see PeopleCode Developer's Guide.

Defining Global Variables

You can also define global variables or objects in the panel PeopleCode before calling an Application Engine program. Only Application Engine PeopleCode actions are able to access the

variables you define, however, the PeopleCode could set a State Record field equal to a number/string variable for use by other Application Engine Actions.

Also, an Application Engine PeopleCode program can read or update a Panel Scroll using a global Rowset Object. When accessing a Panel Scroll from Application Engine PeopleCode, the same rules apply and the same illegal operations are possible that you would see with accessing Record or Panel PeopleCode.

The parameters submitted in a CallAppEngine will be "by value." These parameters "seed" the specified Application Engine State Record field with the corresponding value. If that value is changed within Application Engine by updating the State Record field, the Panel data will not be affected. The only way to update panel buffers or "external" PeopleCode variables from Application Engine is to use global PeopleCode variables and objects.

State Records

Executing PeopleCode from Application Engine Steps allows you to complete some simple operations without having to use SQL. For example, to assign a literal value to an Application Engine State Record field using SQL you may have issued a statement similar to the following:

```
%SELECT (MY_AET.MY_COLUMN)

SELECT 'BUSINESS_UNIT' FROM PS_INSTALLATION
```

You can use a PeopleCode assignment instead.

```
MY_AET.MY_COLUMN = "BUSINESS_UNIT";
```

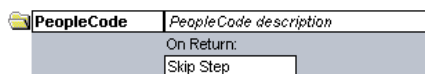
Similarly, you can use a PeopleCode IF statement instead of using a DO When to check the value of a State Record field.

When accessing State Records with PeopleCode, keep the following in mind:

- State Records are records unique to an Application Engine program.
- Within Application Engine PeopleCode, State Record values can be accessed and modified using the standard "record.field" notation.

IF, THEN Logic

From PeopleCode, you can trigger an error status, or false return, by using the EXIT(1) function. Use the **On Return** value on the PeopleCode Action properties to specify how your Application Engine program will behave according to the return of your PeopleCode program.



On Return Action Property

By default, an abort will occur, similar to what happens when a SQL error occurs. But by changing this to *Skip Step* you can control the flow of your Application Engine program.

You can use EXIT() to add an IF condition to a Step or a Section Break. For example,

```
If StateRec.Field1 = 'N'

Exit(1);

Else

/* Do processing */

End-if;
```

Note that the default is Exit(0), so if you don't specify an EXIT or exit value, it's like saying EXIT(0).



Note. You can also specify Exit True or Exit False. And, the default is Exit(0), so if you do not specify an EXIT or Exit value, you are essentially specifying Exit (0).

You must specify a non-zero return value to trigger the On Return action. The concepts of “return 1” and “return True” are equivalent. So, if the return value is non-zero or True, then Application Engine will perform what you specify for On Return, as in Abort or Skip Step. However, if the program returns zero or False, Application Engine ignores the selected On Return value.

Variable Scope

The scope of variables within Application Engine PeopleCode is an important concept to understand. In general terms, it's more important to understand the scope of variables in Application Engine, as a whole, if you think of the State Records as containing the variables used in an Application Engine program.

The following table presents the different types of variables typically used in Application Engine programs and their scope.

<i>Type of variable</i>	<i>Scope</i>	<i>Comments</i>
State Record (Work record)	Transaction (Unit of Work)	Using a work record as your Application Engine State Record means that the values in the Work record cannot be committed. Commits will happen as directed, but any values in Work records are not retained after a commit.
State Record (“Real” record)	Application Engine Program	Using a “real” record as your Application Engine State Record will preserve the values in the State Record on commit, and the committed values are available in the

Type of variable	Scope	Comments
		event of a restart.
Local PeopleCode variables	PeopleCode Program	Local PeopleCode variables are available only for the duration of the PeopleCode program that is using them.
Global PeopleCode variables	Application Engine Program	Global PeopleCode variables are available during the “life” of the program that is currently running. Any global PeopleCode variables are saved when an Application Engine program commits and checkpoints, and therefore they will be available in the event of a restart.
PanelGroup PeopleCode variables	Application Engine program.	Acts like Global variables to Application Engine.

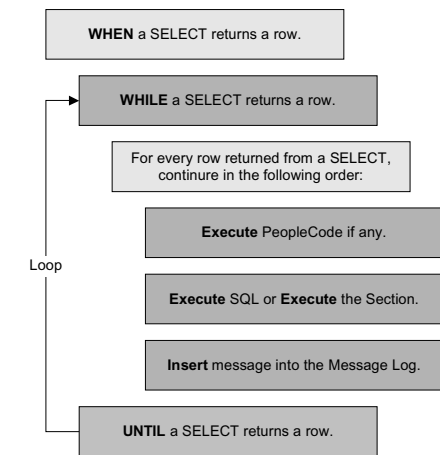
Action Execution Hierarchy

No other types of Actions are required within a Step in conjunction with a PeopleCode Action (or program). So, you can have a Step that contains nothing but one PeopleCode Action. If you include other Actions with your PeopleCode Action within the same Step, it's important to keep in mind the execution hierarchy.

With PeopleCode Actions, Application Engine executes the PeopleCode program *before* the SQL, Call Section, or Log Message Actions, but a PeopleCode program executes *after* any program flow checks.

Since there are multiple Action types, they must execute in agreement within a system, and therefore the order in which Action's execute is significant. At runtime, Actions defined for a given Step are evaluated based on their Action type. All of the Actions types exist within a strict hierarchy of execution. For example, if both a Do When and PeopleCode Action exist within a given Step, the Do When is always executed first.

The following example shows the sequence and level of execution for each type of Action:



Action Execution Hierarchy

Using PeopleCode in Loops

You can insert PeopleCode inside of a “Do” loop, but using PeopleCode inside of high volume “Do” Loops (While, Select, Until) should be made with care. PeopleSoft recommends keeping the number of distinct programs inside the loop to a minimum. As stated previously, you should avoid having PeopleCode performing the actual “work” of the program and instead use it primarily in your Application Engine programs to control the flow (IF, THEN logic), build dynamic SQL, or interact with external systems (using File, Message, or Interlink objects).

Using bind variables instead of literals to pass values to SQL statements is *essential* if the PeopleCode loops or if the PeopleCode gets called in a loop. If the PeopleCode loops, there is a very good chance Application Engine will use a dedicated cursor, which saves the overhead of recompiling the SQL for all iterations. If the PeopleCode gets called from within a loop, Application Engine does not reduce the number of compiles, but Application Engine avoids flooding the SQL cache (for those database servers that support SQL cache) when it uses bind variables. Do not use bind variables for values in a SELECT list or SQL identifiers such as table and column names, as some databases do not support this.



Note. NULL datetime/date/time Bind values are always resolved into literals.

On those database platforms for which PeopleSoft has implemented this feature, Setting BulkMode to TRUE will often result in huge performance gains when inserting rows into a table in a Loop.

In general, avoid having PeopleCode calls within a loop. The “startup and teardown” costs of the PeopleCode interpreter will be multiplied. If you can call the PeopleCode outside of the loop, by all means, use that approach. Not only is it just “good” programming, but also it increases the overall performance with an interpreted language such as PeopleCode.

AESession Object

The AESession PeopleCode object allows you to change the properties of an Application Engine program Section dynamically without having to modify any of the Application Engine tables directly. This allows you to develop “rule-based” applications that conform dynamically to variables that an end user submits through a user friendly panel, such as the Application Engine Request panel.

The AESession object allows developers the following flexibility:

- Portions of SQL determined by checks prior to execution.
- The logic flow conforms as rules change, and the program adjusts to the rules.



For more information regarding the syntax of the AESession object, see PeopleCode Developer's Guide.

When using the AESession object keep the following in mind:

- When you consider using the AESession object, you should first check to make sure that you primarily require dynamic capabilities with the SQL your program generates. Also, make sure that the rules to which your program will conform are relatively static or at least defined well enough such that a standard template could easily accommodate them.
- Explore the possibility of using the SQL Repository to create dynamic SQL for your programs to avoid the complexity of the AESession object using the StoreSQL function
- The AESession Object is only designed to dynamically update SQL-based Actions, not PeopleCode, Call Section, and so on. You can *add* a PeopleCode Action to your generated Section, but you can not alter the PeopleCode.
- The AESession Object is designed for use for online processing. Typically, the dynamic Sections should be constructed in response to an end user action.



Note. Do not call an AESession object from an Application Engine PeopleCode Action.

Synchronous Online Application Engine Calls (CallAppEngine)

In previous releases, you could synchronously make an Application Engine request from your online panels using RemoteCall. Since, Application Engine is no longer a COBOL-based application, this PeopleCode construct is no longer valid to call Application Engine programs. You can still use it to call COBOL programs. If you need to make online calls to an Application Engine program, you need to use the PeopleCode function CallAppEngine.

CallAppEngine provides the ability to call your batch program from an online panel in a synchronous manner. This means the user can't perform another PeopleSoft task until the

Application Engine program completes. Consequently, it's very important to consider the size and performance of the Application Engine program you invoke using CallAppEngine. You need to be sure that the program will run to successful completion consistently within an acceptable amount of time. In short, keep the end user in mind.

If you invoke an Application Engine program using CallAppEngine and the program abends, the end user receives an error, similar to other save time errors, that forces the user to cancel the panel. The CallAppEngine function returns a value based on the result of the Application Engine call. If the program was successful it returns a zero, and if the program was successful, it returns a non-zero.



For more information on the syntax of the CallAppEngine function, see PeopleCode Developer's Guide.

You wouldn't normally invoke your Application Engine programs using CallAppEngine. Rather, you should invoke the bulk of your Application Engine programs through Process Scheduler in an asynchronous manner. Invoking Application Engine programs using CallAppEngine is intended to be the exception, not the norm. We would expect that you would typically use this function if you used RemoteCall() to call an Application Engine program in previous releases.



Note. You *may not* use CallAppEngine to launch one Application Engine program from another. To call another Application Engine program, you need to use the Call Section Action. CallAppEngine is only designed to call Application Engine programs from online panels.

If you issue CallAppEngine from online Panel PeopleCode, in *most* cases you shouldn't be using %TruncateTable or %Execute in any of the Steps in the called Application Engine program. For some platforms, an implicit commit occurs after these statements, and all online processing should be done as a single, *logical* unit of work.

The following sections provide additional details to keep in mind when using the CallAppEngine function.

Events to Trigger CallAppEngine

You need to include the CallAppEngine PeopleCode function within events that allow database updates since if you're calling Application Engine, you're typically intending to perform database updates. This includes the following PeopleCode events:

- SavePreChange (Record)
- SavePostChange (Panel)
- Workflow
- Subscription (Message)

- FieldChange

Process Instance

The Process Instance value is always zero for programs initiated with CallAppEngine. This is because the program called with CallAppEngine runs “in process,” that is; it runs within the same unit of work as the panel group with which it is associated. There is no separate executable involved like when you call a COBOL or SQR program. As long as you delete from all the tables at the end of the Application Engine program, the Process Instance value can remain zero.



Note. There also isn’t any Run Control ID associated with a program invoked by CallAppEngine because that is a batch concept. CallAppEngine applies to online processing.

Process Instances can’t be assigned to Application Engine programs invoked using CallAppEngine for the very reason that it is a single unit of work. To assign a Process Instance, you need to update a one-row table. The update won’t be committed until the very end, so this means all users will be single-threaded through this task.

Your Application Engine program doesn’t perform a commit since it’s part of the panel save. The only commit occurs after your Application Engine program completes and all the Save PeopleCode has finished. As a result, any other users running the same panel simultaneously will have to wait until the first user’s save gets committed. After that, the temporary tables are empty and the next user’s save starts.



Note. If you anticipate scalability problems due to temporary table contention, the solution is to use the multiple temporary table instances feature and reference them dynamically using the %Table meta-SQL construct.



For more information on %Table and using multiple temporary table instances see %Table.

Save Event Details

If you want to execute the Application Engine program based on an end user Save, then use the CallAppEngine function within a Save event. When you use CallAppEngine, you should keep the following items in mind:

- No commits will occur during the entire program run.
- During SavePreChange, any modified rows in the panel have not been written to the database.
- During SavePostChange, the modified rows have been written to the database. The Panel Process issues one commit at the end of the Save cycle.

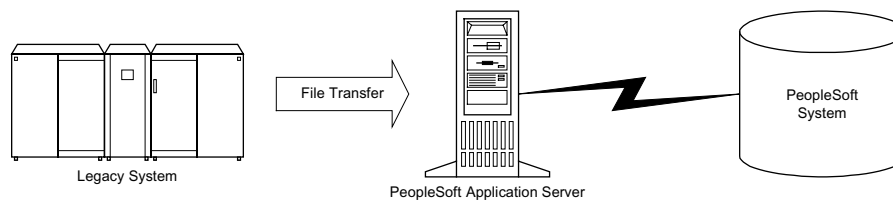
FieldChange Event Details

If you don't want the CallAppEngine call to depend on a Save event, you can also trigger CallAppEngine from a FieldChange event. When having a FieldChange event trigger CallAppEngine, keep the following items in mind:

- No commits will occur within the CallAppEngine. The called program will remain a synchronous execution in the same unit of work.
- The normal FieldChange commit occurs, which frees any locks that the Application Engine program might have acquired.
- For performance reasons, PeopleSoft requires FieldChange PeopleCode that performs a CallAppEngine to be run on the server for three-tier clients. This is accomplished by setting the Run Location of the FieldChange program to be "Application Server", instead of "Default" or "Client." This is set on the PeopleCode Object Properties dialog.
- Do not include a DoSave or DoSaveNow function in the same FieldChange event. This is not allowed, and it indicates that you should be including the CallAppEngine within a Save event.

File Layout Object

The File object allows you to perform file input/output operations with Application Engine using PeopleCode. With the File object, you can open a file (for reading or writing), read data from a file, or write data to it. Using the combination of the File Object and Application Engine provides an effective method to integrate (or exchange) the data stored in a legacy system with your PeopleSoft system. The File Object facilitates the creation of a flat file that both your legacy system and Application Engine programs support.



System Integration using the File Object/Application Engine Combination

In the previous example, the Application Engine program running on the application server uses the File object to read the file sent from the Legacy system and "translate" it so that it can update the affected PeopleSoft application tables. For the PeopleSoft system and the Legacy system to interoperate, you need to first construct a File Object that is compatible for both systems to insert and read data.

We recommend that you attain Rowset and Record access by way of a file using a File Layout definition. You create the File Layout Definition in Application Designer, and it acts as a template for the file that both systems will read from and write to. This greatly simplifies reading, writing and manipulating complex transaction data with PeopleCode.

For the most part, we recommend using the File Object/Application Engine combination in situations where you can't implement the PeopleSoft Application Messaging solution. In some cases, you may opt not to implement our messaging architecture, and in other cases, particular interfaces may not support it. Before taking this approach, PeopleSoft encourages you to explore the possibility of using our messaging architecture first.



For more information on the syntax associated with the File Object, see the PeopleCode Developer's Guide. For more information on creating File Layout definitions, see Application Designer.

Calling COBOL Modules (RemoteCall)

Using the PeopleCode RemoteCall function, you can call, or execute, COBOL modules from a PeopleCode Action. Mainly, PeopleSoft intends this option to support existing Application Engine programs that call COBOL modules. This section is primarily intended for Application Engine developers who are upgrading Application Engine programs from previous releases.

The following sections explain the options involved with Remote Calls from Application Engine programs.

PTPECOBL

Because Application Engine isn't a COBOL based tool (anymore), you can't *directly* call COBOL modules. To make for an easier transition for previous customers, we've added an intermediary, or interface, program called PTPECOBL. This interface program is a PeopleSoft executable that allows you to invoke your called COBOL module and pass it a handful of required values. You code the RemoteCall to invoke PTPECOBL, which, in turn, calls the specified COBOL module.

If you use PTPECOBL, you don't have to write your own executable to handle this task. However, be aware that PTPECOBL does not perform any SQL processing other than retrieve a list of State Record values. Consequently, if your current logic requires prior SQL processing you may want to write your own EXE to call your COBOL module. In most situations, PTPECOBL saves you from having to write a custom EXE to handle each call to a GNT.

PTPECOBL performs the following tasks:

- Initializes the specified State Record in memory.
- Invokes the COBOL module specified in your PeopleCode.
- Submits any required parameters to the called COBOL module.
- Updates the State Record as necessary, issues a commit, and then disconnects from the database after your program completes.



Note. While your COBOL program runs, it can access and return values to the State Record.

To summarize, the following list presents your options for sharing values between the Application Engine program and your called COBOL program:

- **Use State Records.** If you add field names allows you to pass State Record values to the called COBOL program and to get changes passed back to the calling PeopleCode program. If you pass the State Record values in this manner use PTPECACH to retrieve and update values just as PTPEFCNV does.
- **Code custom SQL.** If you do not pass the initial values using State Record fields you need to insert the appropriate SQL in your called COBOL module to retrieve the appropriate values. Then, to return any updated values to the calling Application Engine program, you'll need to insert the appropriate SQL into a PeopleCode program.

If your COBOL program needs values that do not appear in a State Record field, then you can pass PeopleCode variables and values. These variables and values would then be retrieved and/or updated by calling PTPNETRT from within your COBOL program.

- **Create custom EXE.** If you opt to include extra SQL processing and use non-State Record values, for the sake of consistency, it might be a better approach to create a custom EXE to handle your needs. This way you can call your program directly and have it perform all the PTPNETRT processing. Remember that a RemoteCall command can only call an executable program, not a GNT.

When issuing a Remote Call from Application Engine, there are a number of subtle details to keep in mind. Make sure that you read these sections thoroughly.

Syntax and Parameters

The following example shows a sample RemoteCall() issued from an Application Engine PeopleCode Action to a COBOL module.

```
RemoteCall ("PSRCCBL", "PSCOBOLPROG", "PTPECOBL",
"AECOBOLPROG", "MY_GNT",
"STATERECORD", "MY_AET",
"PRCSINST", MY_AET.PROCESS_INSTANCE,
"RETCODE", &RC,
"ERRMSG", &ERR_MSG,
"FIELD1", MY_AET.FIELD1, "FIELD2", MY_AET.FIELD2);
```

The following topics describe each parameter in more detail and the values you can enter.

PSRCCBL

This is the Remote Call “dispatcher.” It executes the specified COBOL program using the connect information of the current operator.

PSCOBOLPROG

Specify the name of the COBOL program to run. In this case it is PTPECOBL.

PTPECOBL

By entering “PTPECOBL” This is the parameter that makes the Remote Call from Application Engine distinct from a normal Remote Call. When you enter this parameter, you, in effect, enable the following parameters, some of which are required.

AECOBOLPROG

Specify the name of the COBOL module you’re calling.

This parameter is required.

STATERECORD

Specify the appropriate State Record that your Application Engine program will share with your COBOL module. PTPECOBL then reserves space in memory for all of the fields on the State Record regardless of whether or not they will ultimately store values for processing.

This parameter is required.

PRCSINST

Specify the State Record and Process Instance field. This retrieves the current Process Instance value that appears on the State Record and submits it to your COBOL module via PTPECOBL.

This parameter is required.

RETCODE, ERRMSG

These parameters are optional. Include RETCODE if you need to return information about any potential problems that the COBOL processing encountered, or use it if your Application Engine program needs to know whether it completed successfully.

Fieldnames and Values

This is where you specify any fields on the State Record that contain initial values for your COBOL module. The quoted field names you specify *must* exist on the specified State Record. The corresponding value can be a PeopleCode variable, a record.field reference or a hardcoded value.



Note. The previous syntax applies only to calling COBOL modules from Application Engine programs.



For more information on the other uses and “normal” syntax for Remote Call, see PeopleCode Developer’s Guide.

Commit

Regarding the important task of committing, keep the following items in mind when using RemoteCall() from an Application Engine program:

- The called COBOL module executes as a separate unit of work.
- We recommend that you execute a commit in the Step immediately preceding the Step containing the RemoteCall PeopleCode Action and also in the Step containing the Remote Call PeopleCode Action. This allows the COBOL process to recognize the data changes made up to the point that it was called, and it also minimizes the time when the process might be in a non-restartable state.
- If you insert *any* SQL processing into your COBOL module, *you* must make sure to commit any updates made by your module. PTPECOBL does not issue any commits.
- If the intent of your COBOL process is to update the value of a passed State Record field, then the calling Application Engine PeopleCode is responsible for ensuring that the State Record field has been modified, and then the Application Engine program is responsible for committing the State Record updates.
- Consider how your COBOL module will react in the event of a restart. Since the work in COBOL will have already been completed and committed will your module ignore a duplicate call or be able to undo/redo the work multiple times? This is similar to issues faced when you execute a Remote call from panel PeopleCode.
- Typically, when a COBOL program updates the database and then disconnects or terminates without having issued an *explicit* commit or rollback, an *implicit* rollback occurs. So, without an explicit commit, the database will not retain any updates.



For more information on exact syntax, returns, and so on for the RemoteCall function, see the PeopleCode Developer’s Guide.

Restrictions

PeopleSoft COBOL support does not include the new features added to the PeopleTools 8 version of Application Engine. So, if you’ve previously developed COBOL programs for use with PeopleSoft, expect the same functionality and limitations for COBOL that existed in prior releases.

The following list presents the relevant COBOL limitations:

- You can only reference fields on a single State Record.

- The State Record used by the COBOL module can't reference sub-records.
- You can't have a Long Character field type on the State Record used by the COBOL module.



Note. Be aware that the COBOL module executes in a *separate* unit of work. Consequently, any database changes made by your COBOL module need to be committed by that program. Otherwise, a “dead lock” may occur between your COBOL module and your Application Engine program.

PeopleTools APIs

You can call all of the PeopleTools APIs, such as Business Components, Trees, Application Messaging, and so on, from an Application Engine program. Keep the following items in mind when using APIs:

- All the PeopleTools APIs contain a Save() method. However, when you call an API from your Application Engine program, regardless of the API's Save() method, the data does not actually get saved until the Application Engine program issues a commit.
- If you've called a Business Component or Tree from an Application Engine program, all the errors related to the API get logged in the PSMessages collection associated with the current session object.
- If you've Published a message, the errors get written to the Message Log and the Application Engine Message Log.
- If an Application Engine program called from Message Subscription PeopleCode encounters errors and the program exits (with Exit (1)), the error is written to the Message Log and will be marked as “Error.”



For more information on the PeopleTools APIs and Error Handling, see PeopleCode Developer's Guide.

CommitWork

The **CommitWork** function commits pending changes (inserts, updates, and deletes) to the database. Keep the following in mind when using CommitWork.

- This function only applies to an Application Engine program that's running in batch (not online). If the program is invoked via CallAppEngine, the **CommitWork** will be ignored. The same is true for commit settings at the Section or Step level.
- This function can only be used in an Application Engine program that has restart *disabled*. If you try to use this function in a program that doesn't have restart disabled, you'll receive a runtime error.

- The CommitWork function is only useful when you are doing row-at-a-time SQL processing in a single PeopleCode program, and you need to commit without exiting the program. In a typical Application Engine program, SQL commands are split between multiple Application Engine Actions that fetch, insert, update, or delete application data. Therefore, you would use the section or step level commit settings to manage the commits.



For more information on the CommitWork function, see PeopleCode Developer's Guide.

Notes on Various PeopleCode Objects and Functions

The following topics provide some examples of common ways that developers can utilize PeopleCode within Application Engine programs.

Do When

Instead of a Do When that checks a %BIND value you can use PeopleCode to perform the equivalent operation. For example, suppose the following SQL exists in your program:

```
%SELECT(EXISTS) SELECT 'Y' FROM PS_INSTALLATION WHERE %BIND(TYPE) = 'X' ,
```

Using PeopleCode, you could insert the following code:

```
If TYPE = 'X' Then

    Exit(0);

Else

    Exit(1);

End-if;
```

If you set the **On Return** parameter on the PeopleCode Action properties to *Skip Step*, this will behave the same as the Do When. The advantage of using PeopleCode lies in the fact that no trip to the database

Dynamic SQL

If you have a select statement that populates a text field with dynamic SQL:

```
%SELECT(AE_WHERE1)

SELECT 'AND ACCOUNTING_DT  <= %Bind(ASOF_DATE) '
```

you can use this PeopleCode:

```
AE_WHERE1 = "AND ACCOUNTING_DT <= %Bind(ASOF_DATE)";
```

Sequence Numbering

If you typically use SELECT statements to increment a sequence number inside of a Do Select, While, or Until loop, you can use the following PeopleCode instead:

```
SEQ_NBR = SEQ_NBR + 1;
```

Using PeopleCode rather than SQL is significant. Since the sequencing task occurs repeatedly inside a loop, the cost of using a SQL statement to increment the counter increases with the volume of transactions your program processes. When you are modifying a program to take advantage of PeopleCode, the areas of logic you should consider are those that start with Steps that are executed inside a loop.



Note. You can also use the meta-SQL constructs %NEXT and %PREVIOUS when performing sequence numbering. Using these constructs may help performance in both PeopleCode and SQL calls.



For more information on Application Engine meta-SQL, see Application Engine Meta-SQL.

Rowsets

You can use Rowsets in your Application Engine PeopleCode. However, keep in mind that using Rowsets means you'll be using PeopleCode to handle more complicated processing, which will degrade performance.

Math Functions

Although, PeopleCode supports a wider range of Math functions with PeopleTools 8, for Application Engine, PeopleSoft recommends using the Math functions that your RDBMS offers when ever possible.

Internally, PeopleCode assigns types to numeric values. Calculations for the Decimal type are processed in arrays to ensure decimal point uniformity across hardware and operating system environments. This is much slower than calculations for type Integer, which are processed at the hardware level.

When PeopleCode converts strings to numeric values, it does so using the internal Decimal type. So, for performance reasons avoid doing calculations using these values.

A third type of numeric value is the Float type. It is not used as frequently for the following reasons:

- Constants are never stored as Float types in the compiled code. For example, 2.5 will always be

Decimal.

- The only way to produce a Float value is by using built-in functions, such as Float() or the Financial math functions.

The Float type is used to produce a Float result only if all operands are also of the Float type. Float operations occur at the hardware level.

To summarize, be aware that PeopleCode does not offer optimum performance when processing non-Integer, non-Float math calculations. If you need to perform calculations with these numeric types, consider allowing the database to perform the calculations in COBOL.

SQL Objects

As opposed to using SQL objects within PeopleCode, PeopleSoft recommends having Application Engine issue the SQL. You may be tempted to use PeopleCode to create a SQL object similar to the following example:

```
&SQL = CreateSQL("SELECT ...");

While(&SQL.FETCH(...))

    <do PeopleCode operation>

End-while;
```



Note. The previous syntax is not intended to be correct; it is merely a hypothetical example.

Rather than coding a PeopleCode program similar to the previous example, PeopleSoft recommends using a Do Select that loops around Sections containing PeopleCode Actions.

It might *appear* easier to code all of the logic within a single PeopleCode program, but splitting the logic into smaller pieces is preferable since you will have better performance, and you get a finer granularity of commit control. You can't cause a commit to occur within a PeopleCode program, but you can issue a commit between Application Engine Steps.



Note. You can't keep global SQL Objects active during a commit, or "across" a checkpoint. PeopleCode behavior does not allow it due to a variety of potential restart issues. If you attempt to access a SQL Object that was open during a checkpoint/commit, you will receive an error. In a complex program containing multiple units of work(multiple commits), you should be using Application Engine SQL Actions with ReUse enabled to manage persistent database cursors.

Arrays

PeopleSoft recommends that you avoid using arrays in your Application Engine PeopleCode. In situations where you are tempted to use an array, explore the use of temporary tables for storing pertinent/affected data. Using temporary tables also lends itself to set-based processing for which Application Engine is most suited.

Set Processing

Although Application Engine achieves high performance while processing row-by-row programs, we believe that the real secret to gaining outstanding performance with Application Engine programs is to employ a technique called set processing. There are circumstances where a row-by-row approach to your batch processing is the only alternative. However, our experience indicates that such circumstances are the exception. As a general rule, we recommend the set processing approach.

In most cases, set processing (if implemented properly) is more efficient than row-by-row processing, especially when you enter the realm of high volume processing. Since Application Engine is designed specifically to process your high volume updates, set processing is vital to the development of the most efficient Application Engine programs. Keep in mind that set processing is not just applicable to Application Engine programs. Whether you're using Application Engine or another batch programming language, proper set processing is the method to use for increased scalability and performance.

Although we evangelize the use of set processing, we have discovered that formal documentation covering this technique is scarce. Perhaps there is an existing SQL Reference guide that explains the concepts of set processing, but as of yet, we have not located it. Consequently, it's our responsibility to provide a foundation in this section for at least understanding what *we* mean by "set processing."

As you begin to design an Application Engine program, we suggest that you consider the volume, or number of rows, the program will be processing and the type of performance required. You should also anticipate any future growth in volume. If the answer indicates that a large volume of rows needs to be processed with optimum performance, you owe it to yourself to explore the benefits of set processing.



Note. By a "large volume of rows" we are referring to thousands of rows or more. Set processing is intended to increase the performance of a program that updates a large number of rows, and it is important to note that you will probably not see much performance gain from set processing if the program is updating very small sets of transactions.

After reading this section you will be familiar with the following:

- What set processing is.
- The advantages of set processing over the row-by-row approach.
- How to think of sets, or columns of data, rather than rows of data.

- Effective set processing.

What is Set Processing?

Set processing is a SQL technique used to process groups, or sets, of rows at one time rather than processing each row individually. Many developers approach a transaction procedurally, in that they follow a repetitive algorithm or loop that selects a row, determines if it meets a given criteria, if so, apply rule *x* to row, update row, commit, select next row, and so on. If you were thinking in terms of set processing, you'd only select, or isolate, those rows that meet the filtering criteria and then run the rule *once* against all the affected rows.

Set processing allows you apply your business rule directly on the data (preferably while it resides in a temporary table) in the database using an UPDATE or INSERT/SELECT statement. The bulk of the performance gain lies in the fact that the processing occurs in the database as opposed to pulling the data into the application program, processing it, and then inserting the results back into the database tables. Since the data never leaves the database with set processing (whether or not it remains in the same table), you effectively eliminate the network round trip and database API overhead required to pull the data into the application program and then insert the results back into the database.



Note. Because the updates in set processing occur within the database, we encourage you to use temporary tables to hold transient data while your program runs. Although temporary tables are not necessarily required for set processing, they are often essential to achieve optimum performance in your batch program.



For more information on the use of temporary tables within set processing, see Temporary Tables.

For example, let's say you have 1,000 employees and you decide to give all of the employees, who are on an hourly pay scale, .25 cent raises. Also, let's say that only a quarter of your staff are paid by the hour.

In a row-by-row approach, the system would perform the following:

- Select each row, or employee, and determine if they meet the criteria (are they hourly?). This is usually done with a WHERE clause in the SELECT to drive the loop.
- If not, go to step 4.
- Run the UPDATE statement, as in, if the criteria is true, add 25 cents to the WAGE *field*.
- Select next row.

The set approach would resemble the following:

```
UPDATE EmployeeTable SET Salary = Salary + 0.25
```

```
WHERE SalaryType = 'HOURLY'
```



Note. The only SELECT statements used in set processing appear in sub-queries of an INSERT or UPDATE statement.

With the row-by-row approach, you run the business rule (Salary increase) 250 times, where as in the set approach the WHERE clause reduces the number rows to only those that are affected (the set), and the program runs the rule once against all the pertinent rows via an UPDATE statement. Also, notice that in the set approach a *column* gets updated as opposed to a *field*. One way to approach set processing is to think in terms of columns rather than rows or fields.



Note. Granted, the previous example is simplistic, the row-by-row method is not optimized, and your business rules are most likely much more complicated than what appears here. We include the example to illustrate that set processing is a different way of applying your business logic. Again, set processing is not unique to Application Engine; you should try to implement it whenever you're using SQL to perform high-volume processing.

Advantages of Set Processing

In most situations that we have encountered with PeopleSoft applications, perhaps 95%, set processing can be implemented to improve performance. This includes those instances in which row-by-row processing seemed like the only alternative. The following list includes some of the major reasons why we encourage you to use set processing and some of the key benefits that you can expect from a set-based program.

- **Improved Performance.** Our internal testing has revealed that, in an overwhelming majority of cases, set processing performs significantly better than its row-by-row counterpart for "reasonable" batch processing volumes. Set-Based programs tend to scale in a geometric manner. Row-by-row processing scales in a more linear manner. When you encounter higher volumes, row-by-row processing can sometimes become overwhelmed.
- **Minimized (PeopleTools) SQL Overhead.** Set Processing is important with Application Engine because Application Engine has built in checkpoint/restart capabilities. Since Application Engine records the status of a program run, there is an avoidable degree of overhead associated with each Application Engine SQL statement that gets executed. If you use set processing, you would tend to use fewer Application Engine SQL Statements that each processed more data than the statements executed in row-by-row processing. As a result, the Application Engine overhead becomes less of a factor in the overall program.
- **Easy Maintenance.** By maintenance, we mean activities that include tuning, fixing, adding enhancements, and so on. Suppose your application logic is in set-based SQL rather than COBOL. If you need to make a fix or add an enhancement to SQL, it's just a matter of modifying the SQL or inserting the new "chunk." With COBOL, you first need to identify all the lines of code affected, modify each line of code, and then re-compile and re-link the program. In short, SQL offers more immediate results.

- **Leveraging the RDBMS.** With set-based processing, you take advantage of the SQL processing engine on the database rather than placing the processing burden and overhead on the application executable. And, as the RDBMS systems get more sophisticated, set-based programs will reap even more performance gains. Using a row-by-row approach and pulling data into the program for each transaction degrades performance regardless of whether the program runs on the batch server or the client. Even if the batch server and database engine are on the same physical machine, the data still travels through a variety of API layers. When a program runs on the client (in row-by-row), you introduce the performance limits imposed by network bandwidth with *each* row of data that the program retrieves from the database.

Using Set Processing Effectively

If you are planning on upgrading older Application Engine programs or developing new ones to adhere to a set-based model, there are a few items to keep in mind or have in place before you begin.

This section contains a collection of topics intended to illustrate some of the items you'll need to have in place before you start coding set-based programs, as well as some tips and guidelines we've accumulated through our own experiences.

SQL Expertise

Even if you're developing row-by-row programs with Application Engine, you should be a SQL expert. With set-based programs, this is especially true. This means knowing SQL inside and out. We recommend that you are particularly keen on the following:

- Group By and Having Clauses
- Complex Joins
- Sub-Queries (correlated and non-correlated)
- Tools for your RDBMS to analyze complex SQL statements for performance analysis

Typically, you'll use these SQL constructs to refine or filter the set to contain only the rows that meet particular criteria. Keep in mind that SQL is what you code with Application Engine, and Application Engine passes that SQL directly to the RDBMS where it gets processed, as is, by the database. If you have a complex SQL statement that works functionally, it may not necessarily perform well if it is not properly tuned.

Planning

As with any development project, well constructed, robust, and efficient Application Engine programs are usually the product of a detailed planning stage where loops, program flow, the use of temporary tables, Sections, Steps, and so on, are discussed.

In an ideal situation, we suggest that you address batch processing as a whole while you are designing the system. Sometimes, systems analysts and developers focus primarily on the online system during the database design, and then they consider the batch component within the

existing database design. Set processing works best in an environment where the data models are optimized for set processing.

For example, you could have a separate staging table for new data that hasn't been processed rather than having numerous cases where existing rows in a table get updated. In set processing, it is much easier to process the data after moving it to a temporary table using an INSERT/SELECT statement rather than just using an UPDATE. A good rule of thumb is to avoid performing updates on "real" application tables, and try to perform your updates on temporary tables. To minimize updating "real" application tables, structure the data model so that it isn't necessary.

Another important consideration is keeping historical data separate from "active" transactions. Once the life cycle of given piece of transaction data is over, so that no more updates are possible, consider moving that data to an archive or history table and deleting it from the "real" transaction table. This will keep the number of rows in the table to a minimum, which improves performance for queries and updates to your "active" data.

In short, designing a system to exploit the benefits of set processing requires a lot of planning up-front, but, for all the time you invest in planning, keep in mind that you'll enjoy the dividends of excellent performance in production mode.

Temporary Tables

Most Application Engine programs require one or more temporary tables to contain transient data. This is especially true of programs that take advantage of set processing techniques. Although temporary tables are not necessarily required for set processing, you will find that well designed temporary tables will complement your set-based program in a variety of ways. For instance, you'll enjoy performance benefits, and if your program runs against a temporary table rather than the master tables, online users and processes can still access the master tables.

Creating temporary tables allows you to achieve one of the main objectives involved with set based processing—the processing remains on the RDBMS server. By storing transient data in temporary tables, you avoid the situation where the batch program fetches the data, row-by-row, and runs the business rule, processes the data, and then passes the updated data back to the database. If the program were running on the client, you'd be taking performance hits due to the network roundtrip and the diminished processing speed of a client (compared to the RDBMS).

Your temporary tables should be designed to accomplish the following:

- Hold transaction data for the current run or iteration of your program.
- Contain only those rows of data affected by the business rule.
- Present key information in a denormalized or "flattened" form, which provides the most efficient processing. (The following paragraphs explain this concept further.)
- Switch the keys for rows coming from the master tables if needed. A transaction may use a different key than what appears on the master tables.

The most efficient temporary tables store the data that a program needs to access in a denormalized or flattened form. Since most programs need to access data that resides in multiple tables, it is more sensible to consolidate all of the affected and related data into one table, the temporary table. It's much more efficient for the program to run directly against the flattened

temporary table rather than relying on the system to materialize complex joins and views to retrieve or update necessary data for each transaction.

If your program requires the use of a complex view to process transactions, then we suggest that you resolve the view into a temporary table for your program to run against. Each join or view that needs to materialize for each transaction consumes system resources and affects performance. In this approach, the system applies the join or view once (during the filtering process), populates the temporary table with the necessary information that the program needs to complete the transaction, and then runs the program against the temporary table as needed.

For example, consider the following situation:

- A program needs to update 10,000 rows on the CUSTOMER table, which contains 100,000 rows of data.
- The CUSTOMER table is keyed by SETID.
- To complete the transaction, the program references data that resides on a related table called PS_SETCNTRL_REC.
- PS_SETCNTRL_REC is used to associate SETID and BUSINESS_UNIT values.
- The transaction is keyed by BUSINESS_UNIT.

Given that set of circumstances, the most efficient processing method would be similar to the following:

- Isolate affected or necessary data from both tables, and insert that into the temporary table. Now, instead of dealing with a 100,000-row CUSTOMER table and a join to a related table, the program faces a 10,000-row temporary table that contains all of the required data to join directly to the transaction data, which can also be in a temporary table. If all necessary columns reside on the temporary tables, the program can modify all the rows at once in a simple UPDATE statement.



Note. The previous bullet item presents two different uses of temporary tables. In one situation, the temporary table is designed to hold setup/control data in a modified form. In the other situation, the temporary table is designed to hold transaction data in a denormalized form, perhaps with additional “work” columns to hold intermediate calculations.

- Make sure the data appears in a denormalized form for optimum processing.
 - Since the transaction is keyed by BUSINESS_UNIT, so should the temporary table that holds the control data. In this case, the table that holds the control data is the CUSTOMER table.
-



For more information and an example of how temporary tables can be used in a set-based program, see Temporary Tables.

Tips

When you sit down to write set-based programs in Application Engine, there are some general guidelines to keep in mind. By "general guidelines" we mean that there are certain techniques or approaches that you will want to keep in mind. On the other hand, there are also a few "techniques" that we've identified as common traps that developers fall into that jeopardize performance gains.

Hybrid Application Programs

A set-based program is not an all-or-nothing situation. As we stated previously, nearly 95% of the business rules you encounter in a PeopleSoft batch program can benefit from set-based processing. There are some rules that call for row-by-row processing, but our experience reveals that these rules are the exception. When an exception arises, we urge you not to let the exception ruin the chances of a "mostly" set-based program. You can have a row-by-row component within a mostly set-based program.

For example, let's say your program contains five rules that you'll run against your data. Four of those rules lend themselves well to a set-based approach while the fifth requires a row-by-row process. In this situation, we suggest you run the four set-based steps or rules first, and then run the row-by-row portion last to resolve the exceptions. Although it's not pure set-based processing, you'll still get better performance than if the entire program used a row-by-row approach.

And, when performing a row-by-row update, reduce the number of rows and the number of columns you select to an absolute minimum to decrease the data transfer time.

For a piece of logic that cannot be coded entirely in set, try to process the bulk of the transactions in set, and handle only the exceptions in a row-by-row loop. A good example of an exception is the sequence numbering of detail lines within a transaction when most transactions only have a single detail line. You can default the sequence number on all the detail lines to 1 in an initial set-based operation, then execute a SELECT to retrieve only the exceptions (duplicates) and update their sequence numbers to 2, 3, and so on.

Avoid the tendency to expand row-by-row processing for more than what's necessary. For example, just because you're touching all of the rows of a given table in a specific row-based process, it doesn't mean that you'll gain in efficiency by running the rest of your logic on that table in a row-based manner.

In the case of updating a table, it's fine to add another column to be set in the UPDATE statement. However, we do not recommend adding another SQL statement to your loop just because your program happens to be looping. If you can apply that SQL in a set-based manner, in most cases, you will achieve better performance with a set-based SQL statement outside the loop.

Filtering

Using SQL, filter the "set" to contain only those rows that are affected or meet the criteria and then run the rule on them. Use the WHERE clause to minimize the number of rows to reflect only the set of affected rows.

Two-Pass Approach

Use a two-pass approach wherein the first pass runs a rule on all of the rows, and the second pass resolves any rows that are exceptions to the rule. For instance, bypass exceptions to the rule during the first pass, and then address the exceptions individually in a row-by-row manner.

Parallel Processes

Carve sets into distinct groups, and then run the appropriate rules or logic against each set in parallel processes. For example, in terms of employee data, you could split the population into distinct sets of "hourly" and "salary," and then you could run the appropriate logic for each set in parallel.

Flat Temporary Tables

"Flatten" your temporary tables. The best temporary tables are de-normalized and follow a flat file model for improved transaction processing.

For example, Payroll control data might be keyed by SET ID and Effective Dates rather than by Business Unit and Accounting Date. Use the temporary table to denormalize the data, and switch the keys to "Business Unit" and "Accounting Date." Afterwards, you can construct a straight join to the Time Clock table, keyed by "Business Unit" and "Date."

Techniques to Avoid

- If you have a series of identical temporary tables, you'll want to re-think your refinement process.
- INSERT/UPDATES from one table to the next during the refinement process.
- Don't attempt to accomplish a task that your RDBMS does not support, as in complex mathematics, non-standard SQL, and complex analytical modeling. Use "standard" SQL for set processing.
- Although sub-queries are a useful tool for refining your set, make sure that you're not using the same one multiple times. If you are using the same sub-query in more than one statement, odds are that you should have denormalized the query results into a temporary table. The key is to identify the sub-queries that appear frequently and, if possible, "flatten" or denormalize the queried data into a temporary table.

Examples of Set Processing

The following topics each contain an example of set processing. It is doubtful that the examples will relate specifically to your unique business needs. However, we hope that the examples will help you begin to see the general approach of set processing so that you can begin implementing set-based programs.

Payroll

In this example, let's say the payroll department needs to give a \$1000 salary increase to everybody whose department made more than \$50,000 profit. The following pseudo code allows you to compare the row-by-row and set-based approaches.

Row-by-Row

```
declare A cursor for select dept_id from department where profit > 50000;

open A;

fetch A into p_dept_id

while sql_status == OK

    update personnel set salary = (salary+1000) where dept_id = p_dept_id;

    fetch A into p_dept_id;

end while;

close A;

free A;
```

Set-Based

```
update personnel set salary = (salary + 1000)

where exists

    (select 'X' from department

     where profit > 50000

     and personnel.dept_id = department.dept_id)
```



Note. The previous example employs a correlated subquery. These play a big role in set-based processing.

Using Temporary Tables

One technique to improve your database performance is to use a temporary table to hold the results of a common sub-query. Effective Dating and Set ID indirection are common types of sub-queries that you can replace with joins to temporary tables. With the joins in place, you can just access the temporary table instead of doing the sub-query multiple times. Not only do most databases prefer joins to sub-queries, but if you can bring multiple sub-queries into a single join as well, the performance benefits can be huge.

In this SETID indirection example, you'll see a join from a transaction table (keyed by BUSINESS_UNIT and ACCOUNTING_DT) to a setup table (keyed by SETID and EFFDT).

To accomplish this using a single SQL statement, you need to bring in PS_SET_CNTRL_REC to map the business unit to a corresponding SETID. This is typically done in a sub-query. You also need to bring in the setup table a second time in a sub-query to get the effective date ($\text{MAX}(\text{EFFDT}) \leq \text{ACCOUNTING_DT}$). If you have a series of similar statements, this can be a performance issue.

The alternative is to use a temporary table that is the equivalent of the setup table. The temporary table would be keyed by BUSINESS_UNIT and ACCOUNTING_DT instead of SETID and EFFDT. You populate it initially by joining in your batch of transactions (presumably also a temporary table) once, as described above, to get all the business units and accounting dates for this batch. From then on, your transaction and setup temporary tables have common keys, which allows a straight join with no sub-queries.

The example in this topic (taken from PeopleSoft Receivables and modified slightly for clarity) may help to illustrate these concepts. The following paragraphs provide some context for the examples.

The original setup table (PS_ITEM_ENTRY_TBL) is keyed by SETID, ENTRY_TYPE and EFFDT.

The de-normalized temporary table version (PS_ITEM_ENTRY_TAO) is keyed by PROCESS_INSTANCE, BUSINESS_UNIT, ENTRY_TYPE and ACCOUNTING_DT, and carries the original keys (SETID and EFFDT) as simple attributes for joining to other related setup tables, as in PS_ITEM_LINES_TBL for this example.

If the program references the setup table in only one INSERT/SELECT or SELECT statement, you wouldn't see increased performance by de-normalizing the temporary table. But, if several SQL statements are typically executed in a single run, all of which join in the same setup table with similar SETID and EFFDT considerations, then the cost of populating the temporary table up front pays off.

Before



Note. The ellipses indicate areas where we've removed SQL for the sake of clarity.

```
INSERT INTO PS_PG_PENDDST_TAO (...)
```

```
SELECT
```

```
. . . . .
```

```
( (I.ENTRY_AMT_BASE - I.VAT_AMT_BASE) * L.DST_LINE_MULTPLR *  
L.DST_LINE_PERCENT / 100 ),
```

```
( (I.ENTRY_AMT - I.VAT_AMT) * L.DST_LINE_MULTPLR * L.DST_LINE_PERCENT / 100 ),
```

```

. . . . .

FROM  PS_PENDING_ITEM I, PS_PG_REQUEST_TAO R, PS_ITEM_LINES_TBL L,
      PS_ITEM_ENTRY_TBL E, PS_SET_CNTRL_REC S, PS_BUS_UNIT_TBL_AR B

. . . . .

WHERE

AND L.ENTRY_REASON = I.ENTRY_REASON

AND L.SETID = E.SETID

AND L.ENTRY_TYPE = E.ENTRY_TYPE

AND L.EFFDT = E.EFFDT

. . . . .

AND E.EFF_STATUS = 'A'

AND S.RECNAME = 'ITEM_ENTRY_TBL'

AND S.SETID = E.SETID

AND S.SETCNTRLVALUE = I.BUSINESS_UNIT

AND E.ENTRY_TYPE = I.ENTRY_TYPE

AND E.EFFDT = ( SELECT MAX(EFFDT) FROM PS_ITEM_ENTRY_TBL Z
                WHERE Z.SETID = E.SETID
                AND Z.ENTRY_TYPE = E.ENTRY_TYPE
                AND Z.EFF_STATUS = 'A'
                AND Z.EFFDT <= I.ACCOUNTING_DT )

AND B.BUSINESS_UNIT = I.BUSINESS_UNIT

/

```

After



Note. The ellipses indicate areas where we've removed SQL for the sake of clarity.

```

INSERT INTO PS_ITEM_ENTRY_TAO

. . . . .

SELECT DISTINCT %BIND(PROCESS_INSTANCE), I.BUSINESS_UNIT, I.ACCOUNTING_DT,

    E.ENTRY_TYPE ...

. . .

FROM   PS_PENDING_ITEM I, PS_PG_REQUEST_TAO R,

        PS_ITEM_ENTRY_TBL E, PS_SET_CNTRL_REC S, PS_BUS_UNIT_TBL_AR B

WHERE  R.PROCESS_INSTANCE = %BIND(PROCESS_INSTANCE)

      AND R.PGG_GROUP_TYPE = 'B'

      AND I.POSTED_FLAG = 'N'

      AND R.GROUP_BU = I.GROUP_BU

      AND R.GROUP_ID = I.GROUP_ID

      AND E.EFF_STATUS = 'A'

      AND S.RECNAME = 'ITEM_ENTRY_TBL'

      AND S.SETID = E.SETID

      AND S.SETCNTRLVALUE = I.BUSINESS_UNIT

      AND E.ENTRY_TYPE = I.ENTRY_TYPE

      AND E.EFFDT = ( SELECT MAX(EFFDT) FROM PS_ITEM_ENTRY_TBL Z

                      WHERE Z.SETID = E.SETID

                      AND Z.ENTRY_TYPE = E.ENTRY_TYPE

                      AND Z.EFF_STATUS = 'A'

                      AND Z.EFFDT <= I.ACCOUNTING_DT )

      AND B.BUSINESS_UNIT = I.BUSINESS_UNIT

/

```

```
INSERT INTO PS_PG_PENDDST_TAO (...)
```

```
SELECT ...
```

```

    ( (I.ENTRY_AMT_BASE - I.VAT_AMT_BASE) * L.DST_LINE_MULTIPLR *
    L.DST_LINE_PERCENT / 100 ),

    ( (I.ENTRY_AMT - I.VAT_AMT) * L.DST_LINE_MULTIPLR * L.DST_LINE_PERCENT / 100 ),

```

```
. . . . .
```

```

FROM   PS_PENDING_ITEM I, PS_PG_REQUEST_TAO R, PS_ITEM_LINES_TBL L,
       PS_ITEM_ENTRY_TAO E

```

```
. . . . .
```

```
WHERE
```

```
. . . . .
```

```
AND L.ENTRY_REASON = I.ENTRY_REASON
```

```
AND L.SETID = E.SETID
```

```
AND L.ENTRY_TYPE = E.ENTRY_TYPE
```

```
AND L.EFFDT = E.EFFDT
```

```
. . . . .
```

```
AND E.BUSINESS_UNIT = I.BUSINESS_UNIT
```

```
AND E.ACCOUNTING_DT = I.ACCOUNTING_DT
```

```
AND E.ENTRY_TYPE = I.ENTRY_TYPE
```

```
/
```

Platform Issues

Set processing does not necessarily behave the same on every database platform. Coincidentally, on some platforms, set processing can encounter performance breakdowns! Some platforms do not optimize UPDATE statements that include sub-queries.

For example, the environments that are accustomed to UPDATES with sub-queries would get all the qualifying DEPT IDs from the Department table, and then, using an index designed by an application developer, update the Personnel table. Other platforms would read through every Employee row in the Personnel table and query the Department table for each row.

On platforms where these types of UPDATES are a problem, try adding some selectivity to the outer query. In the following example, examine the SQL in the "Before" section, and then notice how it is modified in the "After" section to run smoothly on all platforms. You can use this approach to workaround platforms that have difficulty with UPDATES that include sub-queries.



Note. In general, set processing capabilities vary by RDBMS. The performance characteristics of each RDBMS differ with more complex SQL and the set processing constructs discussed in this document. Some RDBMS platforms allow additional set processing constructs that enable you to process even more data in a set-based manner. In cases where performance needs improvement, you will need to tailor or tune the SQL for your environment. The discussion that appears in this document is intended to provide a general foundation for set processing. We assume that you are familiar with the capabilities and limitations of your RDBMS and can recognize, through tracing and performance results, the types of modifications you need to incorporate with the basic set processing constructs we introduce.

Before

```
UPDATE PS_REQ_LINE

SET SOURCE_STATUS = 'I'

WHERE

EXISTS

(SELECT 'X' FROM PS_PO_ITM_STG STG

WHERE

STG.PROCESS_INSTANCE =%BIND (PROCESS_INSTANCE)  AND

STG.PROCESS_INSTANCE =PS_REQ_LINE.PROCESS_INSTANCE AND

STG.STAGE_STATUS = 'I'  AND

STG.BUSINESS_UNIT = PS_REQ_LINE.BUSINESS_UNIT AND

STG.REQ_ID = PS_REQ_LINE.REQ_ID AND
```

```
STG.REQ_LINE_NBR = PS_REQ_LINE.LINE_NBR)
```

After

```
UPDATE PS_REQ_LINE

SET SOURCE_STATUS = 'I'

WHERE

PROCESS_INSTANCE = %BIND(PROCESS_INSTANCE) AND

EXISTS

(SELECT 'X' FROM PS_PO_ITM_STG STG

WHERE

STG.PROCESS_INSTANCE =%BIND(PROCESS_INSTANCE) AND

STG.PROCESS_INSTANCE =PS_REQ_LINE.PROCESS_INSTANCE AND

STG.STAGE_STATUS = 'I' AND

STG.BUSINESS_UNIT = PS_REQ_LINE.BUSINESS_UNIT AND

STG.REQ_ID = PS_REQ_LINE.REQ_ID AND

STG.REQ_LINE_NBR = PS_REQ_LINE.LINE_NBR)
```



Note. This assumes that the transaction table (PS_REQ_LINE) has a PROCESS_INSTANCE column to lock rows that are “in process.” This is another example of designing your database with batch performance and set processing in mind.

This modification allows the system to limit its scan through PS_REQ_LINE to only those rows that the program is currently processing. At the same time, it allows a more “set-friendly” environment to first scan the smaller staging table and then update the larger outer table.

Dynamic SQL

Typically, developers include dynamic constructs in their Application Engine programs to change SQL based on various runtime factors or on user-defined input entered through a panel. There are a variety of ways to include dynamic SQL in your Application Engine programs. For example, you could use:

- Dynamic sections, using the AERSection object.
- Changing SQL using the SQL object.

- References to SQL in your own tables since we support longs.

The AERSection object is primarily designed for online Section building, and therefore won't be most frequently used solution.

PeopleSoft recommends that developers use the SQL Object to store their SQL in the repository. Then if you have a few SQL statements to execute, we suggest generating the SQLIDs based on some methodology, such as a timestamp, and then store these in a table.

When the program runs, your SQL could query this table based on "process" and extract the appropriate SQLIDs to be executed with a SQL Action in a DO SELECT loop.

```
%SQL(%BIND(MY_SQLID, NOQUOTES) )
```

For a "dynamic" Do, the AE_APPLID and the AE_SECTION fields, need to appear on the default State Record.

Debugging Application Engine Programs

As with any program you develop, writing the code represents just half of your development effort. The other half consists of testing your program to make sure it successfully executes the logic you constructed on your whiteboard. With Application Engine you use the Application Engine Trace options and the Application Engine Debugger to monitor the execution of your programs.

In this section, we introduce you to the Application Engine Debugger. After reading this section, you will be familiar with the following items:

- Turning on the Application Engine Debugger
- The options that the debugger offers that allow you to examine your program's execution at a more "granular" level.

Enabling the Application Engine Debugger

When you enable the Application Engine Debugger, you're really just specifying that your program should run in debug mode. Consequently, you need to set this option prior to executing your program. Regardless of which method you use to invoke your Application Engine program, you can still take advantage of the Application Engine Debugger provided that you've enabled it in the appropriate locations.

The following procedure describes the steps you need to complete to have your program run in debug mode.

To debug an Application Engine program

5. Set the Debug Option.

You can set the Debug option in the following locations:

- Launch the Configuration Manager and select the Process Scheduler tab. In the Application Engine group, enable Debug by selecting the Debug checkbox. This is the method applies to all methods of invocation.
- If you used the command line option to invoke your Application Engine program, then you can just include the `-DEBUG Y` parameter in the command line you submit to `PSAE.EXE`. If you already have the Debug flag set in the Configuration Manager, then you do not need to include the `-DEBUG` parameter in your command line.
- If you have PeopleCode in your Application Engine program, you'll want to turn the PeopleCode Debugger on, as well. When you launch your program and the PeopleCode Action executes, you will enter the PeopleCode Debugger at that point.



For more information on enabling the PeopleCode Debugger, see the following procedure.



Note. Setting the debug capabilities in either the Configuration Manager or the command line will turn debug mode on. The only situation where this is not true is when you have Debug enabled in Configuration Manager and you explicitly submit `-DEBUG N` on the command line. In this case, the Configuration Manager setting defines your “default” command line value, and the command line can override the default.



For more information on command line parameters associated with Application Engine, see Command Line.

6. Execute the Application Engine program that you wish to debug.



For more information on executing Application Engine programs, see Invoking Application Engine Programs.

7. At the Application Engine Debugger prompt, enter the appropriate command that enables the desired debugging option.

Each command is represented by a single letter such as ‘X’, ‘L’, or ‘M’. Enter the letter that corresponds to the current option you want to engage. To see a list of the available debugging options, enter `?` at the prompt. For example,

```
PeopleTools 8.0 - Application Engine
```

```
Copyright (c) 1988-1999 PeopleSoft, Inc.
```

```
All Rights Reserved
```

Application Engine Debugger - enter command or type ? for help.

AETESTPROG.MAIN.STATS> ?

Debug Commands:

(Q)uit	Rollback work and end program
E(X)it	Commit work and end program (valid between steps)
(C)ommit	Commit work (valid between steps)
(B)reak	Set or remove a break point
(L)ook	Examine state record fields
(M)odify	Change a state record field
(W)atch	Set or remove a watch field
(S)tep over	Execute current step or action and stop
Step (I)nto	Go inside current step or called section and stop
Step (O)ut of	Execute rest of step or called section and stop
(G)o	Resume execution
(R)un to commit	Resume execution and stop after next commit



For more information on the entire set of Application Engine debugging options see Debugging Options.

To enable the PeopleCode Debugger for Application Engine

1. Sign on to PeopleTools using the same Operator ID that you are going to use to invoke the Application Engine program.

If you do not use the same Operator ID, the two processes can't interoperate.

2. Open Application Designer.
3. Select Debug, PeopleCode Debugger Mode.

Your Application Engine program can be open on the desktop, but you do not need to open the Application Engine program or the PeopleCode Action you want to debug.

4. Select Debug, Break at Start.

This causes the Application Engine program to break prior to executing any PeopleCode programs within it.



For more information on the PeopleCode Debugger, see PeopleCode Developer's Guide.

Debugging Options

The Application Engine Debugger offers the following options for testing the execution of your program. Each option is represented by a single letter that you need to specify at the debugger prompt. To engage the option you select, press ENTER. This section provides a description of each option as well as the corresponding letter command.

Before You Get Started

Here are a few items or tips to keep in mind as you begin debugging your Application Engine programs.

- While using the debugger, you can always enter ? to reveal a list of all the available debugging options.
- In some cases, such as when setting breakpoints or Watch Fields, there are sub-menus that offer more options. Once you are familiar with the commands, you can enter multiple items on the command line to combine commands to bypass the sub-menus. For example, to see a list of the breakpoints, you could enter

```
B L
```

Or, to set a field as a Watch Field, you could enter

```
W S MY_FIELD
```

Or, if it's on a different State Record, enter

```
W S MY_AET.MY_FIELD
```



Note. The exception to this option is Modify, which always presents you the current value and then prompts you to enter the new value. You can, however, enter M MY_AET.MY_FIELD to get directly to the new value prompt.

- The letter commands are not case sensitive. For example, 'Q' or 'q' is a valid command to submit at the prompt.

Quit

Command

Enter Q or q.

Description

The Quit option performs a rollback on the current unit of work in the debugging run, and it ends the debugging session. It effectively acts as an abort of your Application Engine program.

Quit is good for testing Restart. You want to have some of the work committed and some of it uncommitted. Then, abort at that point and rollback the pending work. You want to make sure the program restarts from the point of the last successful commit.

Exit

Command

Enter X or x.

Description

This option is only valid after one Step has completed and another has not already begun. It is not valid once you reach the Action level.

Use this option as an alternative to Quit. Exit will end the program run and the debugging session, but it will also commit the current unit of that the program has already completed. This option can be helpful when testing your Restart logic.

Commit

Command

Enter C or c.

Description

If you want to commit the current unit of work in your program, use this option. It is only valid after a step has completed and before another has already begun. It is not valid once you reach the Action level.

You can use this option, for example, if you want to use your database query tool to check the data in your database.

Break

Command

Enter B or b.

Description

This command allows you to set a breakpoint, which is an arbitrary location that you specify within a program that you are testing or debugging. When the program reaches the specified location you specify, or breakpoint, it will temporarily halt execution to allow you to observe the state of the current process.

With this command you have the following options:

- **Set.** Enter S or s to set a breakpoint location. For example:

```
AETESTPROG.MAIN.TESTNUM> b
(S)et, (U)nset, or (L)ist? s
Program [AETESTPROG]:
Section [MAIN]:
Step [TESTNUM]:

Breakpoint set at AETESTPROG.MAIN.TESTNUM
```

The breakpoint location will default to the current location in the program, but you can specify other Sections or Steps, by overriding the default values that appear in the brackets.

- **Unset.** Enter U or u to remove any breakpoints previously set.

```
AETESTPROG.MAIN.TESTNUM> b
(S)et, (U)nset, or (L)ist? u

Active Breakpoints:

(1) AETESTPROG.MAIN.STATS
(2) AETESTPROG.MAIN.TESTNUM

Remove which breakpoint? 1
```

The debugger presents a list of the Active Break points that you've set. To remove or "unset" a particular breakpoint, enter the corresponding sequence number in the list, such as 1, 2, and so on. In the previous example, the break point set at AETESTPROG.MAIN.STATS will be removed.

- **List.** Enter L or l to view the list of Active Breakpoints. For example,

```
AETESTPROG.MAIN.TESTNUM> b
```

```
(S)et, (U)nset, or (L)ist? l
```

Active Breakpoints:

```
(1) AETESTPROG.MAIN.TESTNUM
```

Since, we unset the breakpoint at AETESTPROG.MAIN.STATS in the previous discussion, there is only one Active Breakpoint now. When you want to enter this command, make sure that you have entered B or b first to specify the Break option. If you just enter L or l from the main command prompt, you will engage the Look option.

Look

Command

Enter L or l.

Description

The Look option allows you to observe the values currently in the State Record associated with the program you are debugging. You need to specify the State Record at the Record Name prompt. By default, the default State Record as specified in your Program Properties will appear with the brackets.

You can also specify a specific field name on the State Record in the Field Name prompt. If you want to look at all the fields in the State Record, leave the '*' within the brackets unchanged. For example,

```
Record Name [AE_TESTAPPL_AET]:
```

```
Field Name [*]:
```

```
AE_TESTAPPL_AET:
```

```
PROCESS_INSTANCE    = 37
AE_PRODUCT           = 'AE'
AE_APPL_ID           = 'TESTAPPL'
AE_STEP              = ' '
AE_SECTION           = ' '
AE_INT_15            = 0
AE_INT_14            = 0
AE_INT_13            = 0
AE_INT_12            = 0
AE_INT_11            = 0
```

```

AE_INT_10          = 0
AE_INT_9           = 0
AE_INT_8           = 0
AE_INT_7           = 8
AE_INT_6           = 11
AE_INT_5           = 3
AE_INT_4           = 0
AE_INT_3           = 0
AE_INT_2           = 0
AE_INT_1           = 0

```

If just want to view the value stored in a specific field of the State Record after a particular Step or Action, just enter the appropriate field name at the Field Name prompt. For example,

```

AETESTPROG.MAIN.TESTNUM> 1

Record Name [AE_TESTAPPL_AET] :

Field Name [*]: AE_INT_6

AE_TESTAPPL_AET:

AE_INT_6          = 11

```

You can also use an asterisk (*) as a wildcard to get a partial list. For example, if you enter

```
AE_INT*
```

at the Field Name prompt, you'll only see the fields that start with AE_INT. This is also true for the Record Name prompt. This is useful both to list multiple fields across multiple records or as a shortcut. If you know there is only one State Record that starts with XXX, you don't have to type the full name—just enter XXX.

Modify

Command

Enter M or m.

Description

This allows you to modify the value of a State Record value for debugging purposes. Suppose the previous Steps did not set a value correctly. However, you may want to see how the rest of the

program would perform *if* the appropriate value existed in the State Record. This allows you to give your program a little help in the debugging or testing phase.

As with the Look command, you need to specify the appropriate State Record (if you are using multiple State Records), and you must specify one field. You can only modify one field at time. For example, if you wanted to set the AE_INT_15 field in the AETESTPROG to 10, you would enter the following:

```
AETESTPROG.MAIN.TESTNUM> m

Record Name [AE_TESTAPPL_AET]:

Field Name [none]: AE_INT_15

Current value:  AE_TESTAPPL_AET.AE_INT_15 = 0

Enter new value (do not use quotes around text strings):

10
```

Using the Look command, you can check to see that the value you specified now exists in the State Record. For example,

```
AETESTPROG.MAIN.TESTNUM> l

Record Name [AE_TESTAPPL_AET]:

Field Name [*]: AE_INT_15

AE_TESTAPPL_AET:

    AE_INT_15          = 10
```

Watch

Command

Enter W or w.

Description

When you specify a field as a Watch Field, the program will stop when the value of the field changes.

Similar to the Break command you have the following options:

- **Set.** Enter S or s to set a Watch field.

```
Set or remove a watch field

AETESTPROG.TESTNUM.INTRESLT> w
```


(S)et, (U)nset, or (L)ist? s

Record Name [AE_TESTAPPL_AET]:

Field Name [none]: AE_INT_7

- **Unset.** Enter U or u to “unset” or remove a Watch field from the list.

AETESTPROG.TESTNUM.INTMSG> w

(S)et, (U)nset, or (L)ist? u

Active Watch Fields:

(1) AE_TESTAPPL_AET.AE_INT_6 = 6

(2) AE_TESTAPPL_AET.AE_INT_7 = 7

Remove watch on which field? 2

To remove a field from the Watch Field list, enter the sequence number in the list that corresponds to the field you want to remove. In the previous example, we will remove AE_INT_7 from the Watch Field list by entering 2.

- **List.** Enter L or l. After the completion of a Step or Action, you can view the values of all the fields that you have included in the Watch list. For example,

AETESTPROG.TESTNUM.INTMSG> w

(S)et, (U)nset, or (L)ist? l

Active Watch Fields:

(1) AE_TESTAPPL_AET.AE_INT_6 = 6

(2) AE_TESTAPPL_AET.AE_INT_7 = 7



Note. You can not set a Watch on a “long text” field.

Step over

Command

Enter S or s.

Description

Use Step Over to execute the current Step to completion and stop at the next Step in the current Section.

The behavior depends on the current “level” or the program. You start at the Step level, and then have the ability to “step into” the Action level. If you are at the Step level and use “step over”, you go to the next Step in the current Section, skipping over All actions (including any Call Sections). If you are at the Action level, “step over” executes the current Action and stops at the next Action in the current step, or at the next step in the current section.

Step Into

Command

Enter I or i.

Description

Use this option to observe a Step or Called Section in a more granular level. For instance, you can check each SQL statement and stop. By using this option and checking the State Record at each stop, you can easily isolate problem SQL or PeopleCode.

As with Step over, the behavior depends on the level. At the Step level, you can “step into” the Action level and stop before the first Action in the Step. At the Action level, if the current Action is a Call Section, Step Into takes you to the first Step in the called Section. For other Action types, Step Into acts the same as Step Over, since there no deeper level in which to step.

Step Out of

Command

Enter O or o.

Description

After you’ve “Stepped Into” a Step or Called Section, use the “Step Out of” option to run the rest of the current Step or Called Section and stop. As with the previous “Step” options, the behavior of Step out of depends on the current level of the program.

At the Step level, Step Out Of completes the remaining Steps in the current Section, returns to the calling Section or Step, and stops at the next Action in that Step. If the Section is MAIN and is not called by another Section or Step, then Step out of behaves the same as the Go option.

At the Action level, Step Out Of completes the current Step and stops at the next Step in the current Section, or if the program is at the end of a Section, Step Out of returns to the calling Section/Step.

Go

Command

Enter G or g.

Description

After the program has stopped at a specific location, and you've examined its current state, you can use the Go command to resume the execution of the program. This is a helpful command when you have breakpoints set. With this command, the program won't stop at a Step or Action; it will only stop at the next breakpoint, watched variable change (Watch Field), or until the program runs to completion.

Run to commit

Command

Enter R or r.

Description

Use this option to resume execution of your program after it has stopped. This command will force the program to stop again after the next commit. This is a good option to use when observing your commit strategy and how it will affect a Restart.

CHAPTER 5

Administration

The previous chapters dealt mainly with the design time aspects of Application Engine. This chapter focuses on the types of administrative or run time tasks involved with your Application Engine programs. These tasks involve deciding how end users will invoke Application Engine programs and monitoring the performance of your programs.

This chapter introduces you to the following topics:

- Invoking Application Engine programs.
- Tracing Application Engine programs.
- Managing Abended programs.
- Restarting programs.

Application Engine Administration Interface

You complete many of the Application Engine tasks in Application Designer, but there are three pages devoted to administration tasks for Application Engine that administrators should be aware of. To navigate to these pages click **PeopleTools, Application Engine**.

The three pages used for Application Engine administration are:

- **Manage Abends.** An interface designed to enable you to deal with Application Engine programs that have ended abnormally and free any locked temporary tables the program may have been using. To access this page select **PeopleTools, Application Engine, Use, Manage Abends**.



For more information on managing abends, see Managing Abends.

- **Temporary Table Usage.** This page allows you to view which programs are associated with which temporary tables within your implementation. To access this page select **PeopleTools, Application Engine, Inquire, Temporary Table Usage**.



For more information on monitoring temporary table usage, see Viewing Temporary Table Usage.

- **Request.** A generic or sample process request page used for invoking Application Engine programs. To access this page select **PeopleTools, Application Engine, Process, Request**.
-



For more information on submitting program requests by way of a page, see Application Engine Process Request Page.

Invoking Application Engine Programs

There are a variety of methods you use to invoke Application Engine programs. Which method you choose depends on your implementation. To invoke or run an Application Engine program, choose one of the following methods:

- Process Scheduler
- Application Engine Process Request Page (which allows you to invoke the program through Process Scheduler with additional parameters).
- PeopleCode
- Command Line

On batch servers, you can launch a program using Process Scheduler or from the command line. On application servers, the Application Engine binaries can be loaded and executed by the application server process request itself or they can be spawned into a separate process.

The most common method to invoke a batch Application Engine program is to use the Process Scheduler. The Process Scheduler is the PeopleTool that is responsible for scheduling and invoking the majority of PeopleSoft batch process requests, including Application Engine programs. However, there are instances where you will need to start an Application Engine program from the command line or by using PeopleCode.

An Application Engine program executes in one of the following modes:

- **Batch.** This is the most typical mode of execution. You invoke programs that run in this mode using Process Scheduler or the Application Engine Process Request Page. Batch mode is also referred to as *asynchronous* execution meaning that it runs independently in the background.
- **Online.** Application Engine programs that execute online, typically get executed from a page with the CallAppEngine PeopleCode function. Such online processes are *synchronous* meaning that subsequent processes wait on the results. For instance, a page may be "frozen" until the online process returns the necessary results. With CallAppEngine there are no COMMITs issued. There is an asynchronous online PeopleCode option, ProcessRequest. With ProcessRequest, COMMITs are allowed.
- **Manual.** To execute an Application Engine program in manual mode, you would use the command line. Usually, you only use this technique during testing or if you need to manually restart the program.

After reading the following sections, you will be familiar with each of the methods with which you can invoke an Application Engine program.

Batch Server Considerations

The following topics describe items to keep in mind concerning running Application Engine programs on your batch server.

Supported Execution Platforms

Application Engine runs on any operating system that PeopleSoft supports as an application server. If your site uses an operating system that is not supported for Application Engine, you need to run your Application Engine programs on the application server.



The only exception to this policy is OS/390 (MVS).

Abnormal Program Ends

Application Engine programs that abend on one batch server should not be restarted on another batch server running a different operating system. This avoids checkpoint data serialization/deserialization issues, such as bit precedence.

Tuxedo Requirements

To run Application Engine programs on the batch server, you need to install Tuxedo. This applies to both UNIX and Windows NT batch servers. If you run your batch server on the same server machine as your application server, then the application server and the batch server can share one Tuxedo install. If your batch server is separate from your application server, then you need to install the Tuxedo CD-ROM to your batch server.

TOOLBINSRV Parameter

The TOOLBINSRV parameter in the Process Scheduler configuration file determines where Process Scheduler invokes an Application Engine program. For high-volume batch environments, PeopleSoft recommends specifying the PS_HOME\bin\server\winx86 directory that exists on the same machine where the Application Engine program runs.

Process Scheduler

In most cases, you will use Process Scheduler to invoke your Application Engine batch programs. This involves creating a Process Definition for each program. Process Scheduler offers a Process Type Definition specifically for Application Engine programs, and the command line and parameters are tailored to invoking Application Engine programs.

Type Definition	Type Definition Options
Process Type: Application Engine Operating System: NT Server Database Type: Microsoft	
Details	
Description:	Application Engine
Generic Process Type:	AppEngine
Command Line:	%%TOOLBINSRV%\psae.exe
Parameter List:	-CT %%DBTYPE%% -CD %%DBNAME%% -CO %%OPRID%% -CP %%OPRPSWD%%
Working Directory:	%%DBBIN%%
Output Destination:	
<input checked="" type="checkbox"/> Restart Enabled	

Application Engine Process Type Definition

Here's the complete parameter list:

```
-ct MICROSOFT -cd %%DBNAME%% -co %%OPRID%% -cp %%OPRPSWD%% -r %%RUNCNTLID%% -i
%%INSTANCE%% -ai %%PRCSNAME%%
```



When creating a Process Definition based on the Application Engine Process Type Definition, the Process Name you assign must *exactly* match your Application Engine program name.

Running Application Engine programs is very similar to running any COBOL or SQR program that you typically invoke with Process Scheduler. Use Application Engine as the generic Process Type Definition, and each Application Engine program that you want to invoke using Process Scheduler requires a unique Process Definition derived from the generic Process Type Definition.

Process Definition	Process Definition Options	Override Options	Destination
Process Type: Application Engine Name: AETESTPROG			
*Description:	Application Engine Test		
Long Description:	This process definition is designed for testing the invocation of Application Engine programs through Process Scheduler.		
*Priority:	Medium		
<input checked="" type="checkbox"/> API Aware <input checked="" type="checkbox"/> Log client request <input type="checkbox"/> SQR Runtime			

Creating a New Application Engine Process Definition



For more information on creating Process Type Definitions and Process Definitions, see Process Scheduler.

Application Engine Process Request Page

You can also invoke an Application Engine program by using the Application Engine Process Request page within the Application Engine Administrative interface. In most cases, this page will only need to be used by developers or system experts who are very familiar with Application Engine and the batch environment on site. When you use the Application Engine Process Request page to invoke an Application Engine program, the program still runs through the Process Scheduler. However, the Application Engine Process Request page allows you to specify additional values and parameters than those that appear within the Process Scheduler Process Definitions.

Keep in mind that, typically, end users will initiate Application Engine programs from an application-specific request page using the Process Scheduler. A systems expert or “power user” may, at times, need to create custom Process Requests that require multiple programs to perform parallel processing or that need to set specific, initial values in a State record. This is when we suggest using the Application Engine Process Request page.



Generally, if “seed” data or other Application Engine request settings are required for a particular program, the application-specific request page will have SQL Execs that do the work transparent to the user. Typically, no end user should be invoking programs from the generic process request page discussed in this section. PeopleSoft uses this page for internal testing and to provide an example of how you can design your program-specific request pages.

After reading the following topics you will be familiar with when it is appropriate to use the Application Engine Process Request page to invoke an Application Engine program, and you will be familiar with the controls on the page.

Overview

The Application Engine Process Request page inserts values into the following tables:

- **AEREQUESTTBL.** Contains all of the values that appear on the page except those in the **Parameters** group.
- **AEREQUESTPARM.** Only includes Initial State record values specified in the **Parameters** group, if needed.



Note. Inserting a row in either of the Application Engine request tables is *not* required to run an Application Engine program. This is a key difference from versions of Application Engine *prior* to PeopleTools 8.x where a row in the Application Engine request tables is required to start a program *regardless* of how it is invoked. The Run Control ID is available to your program via %RunControl, whether or not there’s a row inserted into the AEREQUESTTBL.

You only need to use the Application Engine Request page to invoke Application Engine and insert a row into the Application Engine Request records if you need to perform any of the tasks outlined in the following list:

- Insert initial values into the State record(s) associated with a particular program.
- Set an As Of Date for the Application Engine program to perform retroactive processing.
- Set a non-default market for the program.
- Set up a temporary table image to use if you are submitting a process request that will perform parallel processing.

We use the word “generic” to describe the Application Engine Process Request page because, if you opt to use this method to invoke programs, we encourage you to design custom pages for modifying the Application Engine request tables.

Page Controls

In the Application Engine Administration interface, you will find a “generic” Application Engine Process Request page, as shown in the following example. To navigate to the Application Engine Process Request page, select **Application Engine, Process, Request**.

The screenshot shows the 'Application Engine Request' page. At the top, there are fields for 'User ID' (PTDMO) and 'Run Control ID' (COOP), with a 'Run' button. Below these is the 'Program Name' (AETESTPROG) and a description 'Test application', with a 'Test CallAppEngine' button. A 'Last Run' section displays 'Process Origin: Other', 'Process Instance:', and 'Status: Pending'. Below this are 'Process Frequency' (set to 'Once'), 'Market' (set to 'U.S.'), and 'As Of Date' (11/07/2000). A 'Parameters' section includes 'State Record' and 'Bind Variable Name' fields, with '+' and '-' buttons. At the bottom is a 'Value' field.

Application Engine Process Request Page

- **Last Run.** This group of controls shows you the results from the last time a user submitted a particular Run Control. (Rather than using the Process Monitor)
 - **Process Origin.** This lets you know from where the program was invoked: from the Process Scheduler, Command Line, and so on.
 - **Process Instance.** This is a display-only field for informational purposes. It shows the Process Instance assigned to the previous program run.
 - **Status.** Here you can check the status of the last program run, as in whether it is Successful, Pending, and so on.
- **Process Frequency.** Here you can specify how long a particular Process Request will remain

active or valid.

- **Always.** Select *Always* if you want a Process Request to be run as needed.
- **Once.** Select *Once* if a Process Request is a one-time-only request.
- **Don't.** If you want to disable a Process Request so that no one accidentally invokes it and potentially corrupts data, select *Don't*.
- **Market.** If you want to run a program designed for a non-default Market, select the appropriate Market from the drop-down list.
- **As Of Date.** If you are requesting retroactive processing, specify the appropriate as of date.
- **Parameters.** If you need to set any initial values in State record(s) before a program runs, enter the appropriate values in this group of controls. To add additional rows, press F7.
 - **State Record.** Enter the name of the State Record associated with the program.
 - **Bind Variable Name.** Enter the appropriate field/bind variable for which you are inserting a value.
 - **Value.** Enter the initial value you want to set for the specified field.

Upgrading Request Pages from PeopleTools 7.5

This topic only pertains to situations where you need to upgrade the Application Engine Process Request pages from the previous release.

The following list contains the items you need to keep in mind when updating your process request pages:

- Inserting rows into the PS_AEREQUESTTBL is now optional. It's optional since the program name should match (APPLID) the Process Definition name.
- To invoke a process request from a client, the appropriate security options need to be set in Security Administrator. Specifically, for the Class definition, enable the **Run Client Processes** option in the **Allow Requestor To** group on the Process Profile tab.
- Change the run controls to reference the PS_AEREQUESTTBL instead of the previous PS_AE_REQUEST table.
- Make note of the field changes in the PS_AEREQUESTTBL. For instance, AE_PRODUCT and AE_APPL_ID have been replaced by AE_APPLID. Also, notice the addition of the MARKET and ASOF_DT fields.
- When creating your Process Definition in the Process Scheduler, the Process Name that you specify in the Process Definition *must* match the name of your Application Engine program. Each program now requires it's own Process Definition. When naming the Process Definition, make sure you use the new program name if you picked the default option to change the APPL_ID to have the Product ID concatenated on it during the conversion process.
- You can only invoke one Application Engine program from a particular Process Request page.

Theoretically, you can have users pick different programs from a drop-down list, however, we do not recommend this approach as it introduces a unavoidable degree of potential user error, as in selecting the wrong program to run.

PeopleCode—CallAppEngine

In prior releases, you could invoke Application Engine synchronously from PeopleCode using the RemoteCall function, which is designed to call remote, synchronous COBOL programs. However, Application Engine is no longer written in COBOL, and therefore RemoteCall is no longer valid for Application Engine programs.



Note. The RemoteCall function still applies to calling other COBOL functions; it just no longer applies to Application Engine programs. If you don't convert the RemoteCall PeopleCode that previously called an Application Engine program to use the new function, a Panel Processor error message will appear indicating a process error.

To call a particular Application Engine program synchronously from a page using PeopleCode, you need to use the CallAppEngine() function in your SavePreChange or SavePostChange PeopleCode. The basic syntax for CallAppEngine() is as follows:

```
CallAppEngine(applid [, statereclist ]);
```



For more information on the syntax, returns, and parameters involved with the CallAppEngine() function, refer to PeopleCode Developer's Guide.

The following list of guidelines helps you to determine when it is appropriate to use the CallAppEngine() function.

- Use CallAppEngine if the program you are invoking is a “quick” process or will not require a user to wait for an unreasonable amount of time for the transaction to complete. Keep in mind, this is a synchronous process, which means the end user must wait for any process invoked by CallAppEngine() to complete before doing *anything* else. Developers need to consider the end user. If the called program will cause an unreasonable delay, then we suggest using another alternative, such as the ScheduleProcess PeopleCode function.
- Use CallAppEngine when you have a complex, SQL intensive business process that must run in batch *and* online or the process requires the use of dedicated temporary tables. If this is not the case, you are usually better off writing the entire program in native PeopleCode. Keep in mind that if you've written logic in PeopleCode, presumably for online, and you want to re-use it in batch, you may be forced into row-by-row processing. We recommend designing the batch logic *first*, and then you can decide whether to have a separate online version or just re-use the batch code—using CallAppEngine. Developers need to consider the tradeoff between code re-use and performance. It is inherently more difficult (but not impossible) to develop a common solution that will perform adequately in both batch and online.
- Do not use CallAppEngine within an Application Engine PeopleCode Step. If you need to call

an Application Engine program from another Application Engine program, you must use the Call Section Action.

- Do not use CallAppEngine if you need to control the COMMIT. Programs called with CallAppEngine are embedded within a larger unit of work defined by the panel trigger, such as a page save.



Note. Online PeopleCode that calls CallAppEngine should be set to run on the application server. You will encounter performance issues if you attempt to run PeopleCode on the Client in a three-tier configuration, because every SQL statement that Application Engine issues needs to be serialized and then sent to the application server for execution.



For more information on the Call Section Action, see Call Section Actions.

Command Line

In some cases, you may want to invoke an Application Engine program through the command line. For instance, this method is typically used in the following situations:

- **Restarting.** When a program abends, a system administrator might restart the program using the command line. If needed, you can locate all of the specific program/process information from the Process Monitor on the Process Request Detail dialog. Normally, end users (or system administrators) will perform a Restart from the Process Monitor.
- **Development/Testing.** Many developers include the command line in a batch file to launch a program they are developing or testing. This way, they can quickly execute the batch file as needed. This also allows separation of development of the Application Program from its associated pages.
- **Debugging.** To debug a program with a Run Location of Server, you can log into the server (using Telnet, for example) and invoke the program from the command line.

To invoke an Application Engine program from the command line, you need to specify the Application Engine executable (PSAE.EXE) followed by the required parameters, as shown in the following example:

```
psae -CT <dbtype>  
  
      -CS <server>  
  
      -CD <database name>  
  
      -CO <oprid>  
  
      -CP <oprpswd>  
  
      -R  <run control id>
```

```

-AI <program id>

-I <process instance>

-DEBUG <Y|N>

-DR <Y|N>

-TRACE <value>

-DBFLAGS <value>

-TOOLSTRACESQL <value>

-TOOLSTRACEPC <value>

```

Or, if your command line parameters are stored in a text file, you can enter:

```
psae <parmfile>
```



For Windows NT and UNIX servers, you need to set the PS_SERVER_CFG environment variable before you invoke an Application Engine program from the command line. PS_SERVER_CFG must contain the fully qualified name of a correctly configured Process Scheduler PSPRCS.CFG file. When Application Engine runs from the command line, it resolves %PS_SERVDIR% to the value of the environment variable PS_SERVDIR instead of the parent directory of a Process Scheduler configuration.

The following table contains the command line parameters associated with the PSAE.EXE, and shows you which parameters are required.

Parameter	Description and Values
-CT <dbtype>	Required. Corresponds to the type of the database to which you are connecting. Valid values are MICROSOFT, ORACLE, SYBASE, INFORMIX, DB2UNIX, and DB2.
-CS <server>	Required for Sybase, Informix. For those platforms that require a server name as part of signon, enter the appropriate server name. This affects Sybase, Informix, and Microsoft SQL Server. However, for Microsoft SQL Server, this option is <i>valid</i> but not required.
-CD <database name>	Required. Enter the name of the database to which the program will connect.
-CO <oprid>	Required. Enter the Operator ID that is running the program.
-CP <oprpswd>	Required. Enter the password associated with the specified Operator ID/User ID.

Parameter	Description and Values
-R <run control id>	Required. Enter the Run Control ID to use for this run of the program.
-AI <application id>	Required. Specify the Application Engine program to run.
-I <process_instance>	Required for Restart. Enter the Process Instance for the program run. The default is 0, which means Application Engine uses the next available Process Instance.
-DEBUG <Y N>	This parameter controls the Debug utility. Enter <i>Y</i> to indicate that you want the program to run in debugging mode, or enter <i>N</i> to indicate that you do not.
-DR <Y N>	This parameter controls Restart. Enter <i>Y</i> to disable Restart, or enter <i>N</i> to enable Restart. (DR represents Disable Restart).
-TRACE <value>	<p>This parameter turns on the Application Engine trace. To enable tracing from the command line, enter this parameter and a specific Trace value. The value you enter is the sum of the specific traces that you want to enable. The current traces and values are:</p> <p>1—Initiates the Application Engine Step trace.</p> <p>2—Initiates the Application Engine SQL trace.</p> <p>128—Initiates the Application Engine timings file trace, which is similar to the COBOL timings trace.</p> <p>256—Includes the PeopleCode Detail Timings in the 128 trace.</p> <p>1024—Initiates the Application Engine timings table trace, that stores the results in database tables.</p> <p>2048—Initiates the database optimizer “explain”, writing the results to the trace file. This option is supported only on Oracle, Informix, and Microsoft SQL Server.</p> <p>4096—Initiates the database optimizer “explain”, storing the results in the Explain Plan Table of the current database. This option is supported only on Oracle, DB2, and Microsoft SQL Server.</p> <p>So, if you wanted to enable the 1, 2, and 128 traces, enter 131—the sum of 1, 2, and 128. To indicate that you do not want any traces, enter 0. If you don’t explicitly enter 0, Application Engine will use the trace value set in the Configuration Manager.</p>
-DBFLAGS	<p>Using this parameter you can disable the %UpdateStats meta-SQL construct.</p> <p>To disable %UpdateStats enter:</p>

Parameter	Description and Values
	-DBFLAGS 1
-TOOLSTRACESQL <value>	Use this parameter to enable the SQL Trace.
-TOOLSTRACEPC <value>	Use this parameter to enable the PeopleCode Trace.
psae <Parmfile>	<p>If you submit a file to Application Engine as the first parameter in the command line, Application Engine reads the contents of the file and interprets the contents as if it were parameters entered on the command line. This option is intended mainly for the Windows NT or UNIX Process Scheduler Server environment. For example,</p> <pre>psae \$temp/myparmfile.txt</pre> <hr/> <p>Note. For security reasons, after Application Engine interprets the contents of the <parmfile>, it immediately deletes the <parmfile>.</p> <hr/>



For more information on trace values see Trace.

Application Engine Server Caching

Application Engine caches meta-data, just like the application server. This enhances performance because the program can refer to the local cache for any objects that it uses frequently rather than performing costly network trips to the database.

CacheBaseDir

Application Engine programs that run on the NT or UNIX server each lock their own cache directory for the duration of the run. These directories are found under the master CACHE directory. The master directory is created under the directory specified by the CacheBaseDir variable in the Process Scheduler configuration file. If all existing cache directories are locked, a new one is created. Cache sub-directories are named sequentially, starting at 1.



If you are using Windows Clients and running Application Engine programs on the workstation, Application Engine uses the cache directory specified in the Configuration Manager.

If you do not enter a fully qualified path for the CacheBaseDir variable, then Application Engine creates the cache directory within the directory in which the program is set to run.



Do not share the CacheBaseDir with application servers, and do not use environment variables when specifying CacheBaseDir, because the system does not resolve them. For example, do not have CacheBaseDir=\$PS_HOME.

Server Caching

In the PSPRCS.CFG (PS_SERVER_CFG), there are two additional cache parameters. They are:

- Enable Server Caching.
- Server Cache Mode.

PeopleSoft recommends that you *do not* alter these settings from the delivered defaults. These settings are reserved for future use.

Tracing Application Engine Programs

After you've developed your Application Engine program and tested it in a development environment to make sure it works as expected, you can start to focus on the program's performance. Ideally, with batch programs, you want the program to process as many rows of data in the least amount of time. Although there is an unavoidable degree of overhead associated with the execution of any program, by using techniques like set processing and tuning your SQL you can improve the efficiency of your programs. One way to gauge the efficiency of your program is to take advantage of the variety of trace options that are specific to Application Engine.

There are a variety of traces that you can use to monitor the performance of your Application Engine programs. The following list presents your tracing options that are specific to Application Engine:

- Application Engine Step and SQL trace
- Application Engine Statement Timings trace
- PeopleCode Detail Timings trace
- Database Optimizer trace



The general PeopleTools SQL and PeopleCode traces also apply to Application Engine programs.



For more information on the PeopleTools SQL and PeopleCode traces, see [Trace](#).

You can use the Application Engine tracing functionality to monitor the execution of your Steps, your SQL, the Timings (for PeopleCode and SQL), and how the optimizer for your database is handling your SQL constructs within your program.

After reading this section, you will be familiar with the following:

- How to turn on Application Engine traces.
- Where to look for the trace results.
- What sort of information you can expect to find within a particular Application Engine trace.

Enabling Application Engine Traces

By default, all Application Engine traces are turned off. When you need to see a trace or a combination of traces, you need to set the desired trace options prior to executing the program.

You enable, or turn on, traces using the following methods:

- Command Line.
- Server Configuration Files.
- Configuration Manager.

The following sections describe how you would enable the Application Engine traces using each of these methods.

Command Line

This command line option is available on Windows NT and UNIX. This option is not available when calling Application Engine programs from PeopleCode.

To enable tracing from the command line, you need to include the `-TRACE` parameter within the command line that you submit to `PSAE.EXE`. You include the `-TRACE` parameter in addition to the other required parameters. For example,

```
n:\pt810\bin\client\winx86\psae.exe -CT MICROSOFT -CD PT800GES -CO PTDMO -CP  
PTDMO -R PT8GES -AI AETESTPROG -I 45 -TRACE 2
```



For more information on the command line parameters associated with Application Engine, see [Command Line](#).

The Application Engine traces and their corresponding values are:

Value	Trace Description
1	Initiates the Application Engine Step trace.
2	Initiates the Application Engine SQL trace.
128	Initiates the Statement Timings trace to file, which is similar to the COBOL timings trace to file.
256	Adds Initiates the PeopleCode Detail to the file for the Timings trace.
1024	Initiates the Statement Timings trace, but, instead of writing to the trace file, this trace stores the results in the following tables: PS_BAT_TIMINGS_LOG and PS_BAT_TIMINGS_DTL.
2048	Adding this value requests a database optimizer trace file.
4096	Requests a database optimizer to be inserted in the Explain Plan Table of the current database.

If you use the command line, you'll need to make sure that you enter the combination of options you desire. This is similar to adding the trace values using PSADMIN, such as the COBOL statement timings or the SQL statement trace value. To specify a combination of the traces, just enter the sum of the corresponding trace values. For example, if you wanted to enable the Step (1), the SQL (2), and the Statement Timings (128) traces, you would enter 131—the sum of 1, 2, and 128.

If you want to completely disable tracing, you need to explicitly specify `-TRACE 0`. If you don't include the `-TRACE` flag in the command line, Application Engine will use the value specified in the Process Scheduler configuration file or the Configuration Manager. Otherwise, the command line parameters override any trace settings that may be set in the Configuration Manager.

Server Configuration Files

You can also turn on traces in the configuration files for both the application server and the Process Scheduler server. To do so, you set the TraceAE parameter in the Trace section to the desired level of tracing. You can use PSADMIN to set this parameter.

Process Scheduler Server

In the Process Scheduler configuration file, you can set the TraceAE parameter in the Trace section to the desired level of tracing. You can use PSADMIN to set this parameter.

Values for config section - Trace

TraceFile=%PS_SERVDIR%\logs\PeopleTools.trc

TraceSQL=0

TracePC=0

TraceAE=<add trace value>

This option is available on Windows NT and UNIX, and this option *only* applies to Application Engine programs invoked in batch mode.



The TraceFile parameter does not specify the location of the Application Engine trace file; it only applies to the generic PeopleTools SQL and PeopleCode traces.

Application Server

For programs invoked by PeopleCode and run on the Application Server, you set the TraceAE parameter in the Trace section to the desired value in the Application Server configuration file (PSAPPSRV.CFG). You can use PSADMIN to set this parameter.

Values for config section - Trace

TraceSql=0

TraceSqlMask=12319

TracePC=0

TracePCMask=4095

TracePPR=0

TracePPRMask=4095

TraceAE=<add trace value>

This option is available on Windows NT and UNIX.

Configuration Manager

For processes running on a Windows workstation, you can set your trace options on the Configuration Manager. The Application Engine trace options reside on the Trace tab. This procedure is only valid if you are running Application Engine programs on a Windows workstation—the development environment.

To set Application Engine Traces

1. Launch the Configuration Manager, and select the Trace tab.

Look for a group of checkboxes called **Application Engine Trace**. This group box contains the traces that are unique to Application Engine.

2. Select the trace options that apply to your current situation.

You can select any combination of the options.

3. After you have selected the appropriate options, remember to press either the **Apply** or **OK** button to set your trace options.

Locating the Trace File

Where you look for the generated trace file depends on how you invoked the program and the operating system on which the program runs.

If the program runs on the Windows workstation, look for the trace file in %TEMP%\PS\<db name>.

If you launched the program from PeopleCode, look for the trace file %TEMP%\PS\<db name> on NT and \$PS_HOME/log/\<db name>.

If you launched the program from the command line, look for the trace file in the directory specified in the Log/Output field in the PS_SERVER_CFG file.

If you launched the program from Process Scheduler, look for the trace file in a sub-directory of the directory specified in the Log/Output field in the PS_SERVER_CFG file.

If you run the program without specifying a Process Instance, Application Engine names the file according to the following convention.

AE_<Date/Time_Stamp>_<OS_PID>.AET

Where the date/time stamp is <mmddhhmmss>.

If you run the program with a Process Instance specified, Application Engine names the file according to the following convention.

AE_<Program_name>_<Process_Instance>.AET



For an Application Engine program running on a server, PeopleTools writes the generic PeopleTools trace for SQL and PeopleCode trace files to the same directories as the AET traces. The prefix of the trace file name is also the same, and the suffix is trc. On the Windows workstation, the trace is written to the “People Tools Trace File” specified in the Trace folder of the Configuration Manager.

Understanding the Trace Results

The following topics briefly describe what you can expect to see within each Application Engine trace option. Keep in mind that these are just sample trace files of a simple demonstration program.

At the top of each trace there is useful information that helps you to identify the PeopleTools version, the database name, and the RDBMS type. In an environment with multiple databases and database types, this proves to reduce confusion. For example,

```
PeopleTools 8.10 -- Application Engine

Copyright (c) 1988-2000 PeopleSoft, Inc.

All Rights Reserved

Database: PT810GES(SqlServer)
```

Step

The Step trace reports each step name that your program executes and in what order. Associated with each Step you'll see a timestamp, the DO level, and the Action type.

Also, the trace shows the Steps that execute within a called Section by the indented formatting. For example, a Step that executes within a called Section is preceded by two periods (..) while other Steps are preceded by only one period. The following is a sample trace file.

```
PeopleTools 8.10 -- Application Engine

Copyright (c) 1988-2000 PeopleSoft, Inc.

All Rights Reserved

Database: PT810GES(SqlServer)


16.42.29 2000-05-12 Tracing Application Engine program AETESTPROG Test
application

16.42.29 .(AETESTPROG.MAIN.STATS) (Do Select)

16.42.29 .(AETESTPROG.MAIN.STATS) (Call Section AETESTPROG.STATS)

16.42.30 ..(AETESTPROG.STATS.SECTIONS) (SQL)

16.42.30 ..(AETESTPROG.STATS.STEPS) (SQL)

16.42.30 ..(AETESTPROG.STATS.STMTS) (SQL)

16.42.30 ..(AETESTPROG.STATS.MSG) (Log Message)

16.42.31 .(AETESTPROG.MAIN.STATS) (Do Fetch)

16.42.31 .(AETESTPROG.MAIN.STATS) (Call Section AETESTPROG.STATS)

16.42.31 ..(AETESTPROG.STATS.SECTIONS) (SQL)

16.42.31 ..(AETESTPROG.STATS.STEPS) (SQL)

16.42.31 ..(AETESTPROG.STATS.STMTS) (SQL)
```

```

16.42.31 ..(AETESTPROG.STATS.MSG) (Log Message)

16.42.31 ..(AETESTPROG.MAIN.STATS) (Do Fetch)

16.42.31 ..(AETESTPROG.MAIN.TESTNUM) (Call Section AETESTPROG.TESTNUM)

16.42.31 ..(AETESTPROG.TESTNUM.INTEGER) (SQL)

16.42.31 ..(AETESTPROG.TESTNUM.INTRESLT) (SQL)

16.42.31 ..(AETESTPROG.TESTNUM.INTMSG) (Log Message)

16.42.31 ..(AETESTPROG.MAIN.DBSTATS) (Call Section AETESTPROG.DBSTATS)

16.42.32 ..(AETESTPROG.DBSTATS.UPDSTATS) (SQL)

16.42.32 ..(AETESTPROG.MAIN.PCODE) (Call Section AETESTPROG.PCODE)

16.42.32 ..(AETESTPROG.PCODE.PCode01) (PeopleCode)

16.42.32 ..(AETESTPROG.PCODE.PCode02) (PeopleCode)

16.42.33 Application Engine program AETESTPROG ended normally

16.42.33 Application Engine ended normally

```

SQL

The SQL trace shows the formatted SQL processes including COMMITs, ROLLBACKs, and Restarts. You can also view the Buffers associated with each SQL statement. Using the SQL trace to spot errors in your SQL and to view your COMMIT strategy.

The following is a sample of the information contained in the SQL trace file.

```

PeopleTools 8.10 -- Application Engine

Copyright (c) 1988-2000 PeopleSoft, Inc.

All Rights Reserved

Database: PT810GES(SqlServer)

%Select(AE_APPL_ID, AE_PRODUCT)  SELECT AE_APPL_ID, AE_PRODUCT FROM

PS_AE_APPL_TBL ORDER BY AE_APPL_ID, AE_PRODUCT

/

-- Buffers:

--      1) EC_AUDIT

--      2) PO

```

```

%Select (AE_INT_5)    SELECT COUNT(*) FROM PS_AE_SECTION_TBL WHERE AE_PRODUCT =
:1    AND AE_APPL_ID = :2
/
-- Buffers:
--    1) 1

COMMIT

/

%Select (AE_INT_6)    SELECT COUNT(*) FROM PS_AE_STEP_TBL WHERE AE_PRODUCT =
'PO'    AND AE_APPL_ID = 'EC_AUDIT'
/
-- Buffers:
--    1) 7

COMMIT

/

%Select (AE_INT_7)    SELECT COUNT(*) FROM PS_AE_STMT_TBL WHERE AE_PRODUCT =
'PO'    AND AE_APPL_ID = 'EC_AUDIT'
/
-- Buffers:
--    1) 7

COMMIT

/

```

Application Engine Statement Timings

The Application Engine Statement Timings trace is very similar to a COBOL Timings trace in which you monitor the execution of your COBOL programs for performance evaluations. The Statement Timings trace as well as the DB Optimizer trace are intended primarily so that you can

gather performance information that you can use to determine bottlenecks or particular areas of a program where performance is impeded. Once bottlenecks have been identified, it may be possible to modify your program to run more efficiently, or you may want to change the database schema and configuration to optimize the execution of your program.

The statement timings trace is invaluable for tuning an Application Engine program. It may also be useful as a default trace level for all production runs to provide a metric for long term performance trends.

By examining all of the figures in this trace, you can easily identify areas of your program that are not running as efficiently as possible. For instance, if Compile counts are high, you can reduce the numbers by using Application Engine's ReUse feature. If INSERTS appear to be running slow and you have many of them, you can increase the performance by using Application Engine's Bulk Insert feature.

You can opt to write this trace to a file, or you can write the results to tables. Either way, timings trace overhead is minimal. Internal testing reveals that the Application Engine trace has an overhead between 2% and 5% of total runtime.

Each value in the trace, including cumulative totals, appears in a form rounded to the nearest tenth of a second, but totals are calculated using non-rounded timings.



Application Engine does not write the timings trace to table for programs invoked by CallAppEngine. To write to the table a Process Instance is required, and programs invoked by CallAppEngine are not assigned a Process Instance.

The following sections describe the File and the Table options available with the Statement Timings trace.

File

In the following discussion, you will find the generated trace file divided into the different sections that appear in the trace file. Each section is described individually.

SQL Counts and Timings (Section 1)

The first section of the trace file is the SQL section. It records the performance of application specific SQL. The trace values appear within a series of columns and sections as shown in the following example.

	C o m p i l e		E x e c u t e		F e t c h		Total
SQL Statement	Count	Time	Count	Time	Count	Time	Time

PeopleCode							
BI_RECFIELD_SEL			1	0.0	153	0.0	0.0

INSERT PS_INTFC_BI_NTMP1	1	0.0	0	0.0	0.0	0.0
--------------------------	---	-----	---	-----	-----	-----

<snip>

UPDATE PS_INTFC_BI_HTMP1	7	0.0	0	0.0	0.0	0.0
--------------------------	---	-----	---	-----	-----	-----

0.2

Application Engine

COMMIT	0	0.0	76	0.2	0	0.0	0.2
--------	---	-----	----	-----	---	-----	-----

LOGMSG	0	0.0	4	0.1	0	0.0	0.1
--------	---	-----	---	-----	---	-----	-----

0.3

AE Program: BIIF000

AE.UPD_AE.S	1	0.0	16	0.0	0	0.0	0.0
-------------	---	-----	----	-----	---	-----	-----

ASGNIVC1.NTMP.D	1	0.0	1	0.0	2	0.0	0.0
-----------------	---	-----	---	-----	---	-----	-----

ASGNIVC1.REGISTER.D	1	0.0	1	0.0	2	0.0	0.0
---------------------	---	-----	---	-----	---	-----	-----

ASGNIVC2.UPDIVC.D	1	0.0	1	0.0	8	0.0	0.0
-------------------	---	-----	---	-----	---	-----	-----

ASGNIVC5.CK_BIHDR.D	1	0.0	1	0.0	1	0.0	0.0
---------------------	---	-----	---	-----	---	-----	-----

ASGNIVC5.INSHTMP.S 0.1 (BulkInsert)	1	0.1	17	0.0	1	0.0	
----------------------------------------	---	-----	----	-----	---	-----	--

ASGNIVC6.Step01.S	1	0.0	16	0.0	0	0.0	0.0
-------------------	---	-----	----	-----	---	-----	-----

<snip>

UPIVCAMT.UPCHDRAM.D	7	0.4	7	0.0	7	0.0	0.4
---------------------	---	-----	---	-----	---	-----	-----

UPIVCAMT.UPHDRAMT.S	7	0.8	7	0.0	0	0.0	0.8
---------------------	---	-----	---	-----	---	-----	-----

41.6

The following table contains more information about each column within the first section of the trace file.

Column/Section	Description
SQL Statement	Application Engine SQL Actions and stored SQL objects always have a statement ID. The SQL Statement column shows the statement ID so that you can attribute the trace values to individual SQL statements. In the case of SQLExec SQL, a snippet of the SQL statement appears in the first column to help you identify it. For SQL objects, we recommend that you use the TraceName property in the CreateSQL so that you can uniquely identify it in the traces.
Compile Column	This column of values shows how many times the system compiled a SQL statement and how long it took. "Compiled" refers to the SQL statement being sent to the database to be parsed and optimized, and it also includes the time required for the first resolution of any PeopleSoft Meta SQL.
Execute Column	The Execute column shows how many times the system executed the SQL statement and the time consumed doing so. "Executed" refers to the system sending the compiled SQL to the database server to be run against the database.
Fetch Column	The Fetch column applies to SELECT statements. It shows how many rows your program fetched from the database and how much time this consumed. Keep in mind that the system must first Execute a SELECT statement against the database to find the relevant rows and generate an "active" set. Once the set exists, the program must still fetch the rows. Some database API's have buffered fetches which means that the fetch may include more than one row. This makes subsequent fetches "free" until the buffer becomes empty.
Total Column	The Total column shows the sum of the Compile, Execute and Fetch times of the SQL statement. Some database API's may defer a Compile to the Execute and/or defer an Execute to the first Fetch.
PeopleCode SQL	This sub-section is for SQL executed from PeopleCode Actions. Compile counts and times for such SQL is included in Execute Count and Times because the programmer does not explicitly control ReUse. To determine whether ReUse is occurring, you will need to do a program run after enabling the generic PeopleTools trace for SQL statements, API calls, and so on. As a starting point, PeopleSoft suggests a trace value of 31 as a starting point.

Column/Section	Description
Application Engine SQL	<p>This sub-section reveals the time attributed to Application Engine overhead that is not directly related to the SQL within your program. For example, the values in this section represent the SQL generated for checkpoints, commits, and so on. If there are COMMITs without checkpoints, it indicates that restart has been disabled, or a restartable program has called a non-restartable program.</p> <p>If the time consumed performing a checkpoint or committing seems more than expected, you should try to reduce it if possible by setting the Commit Frequency of the Steps containing DO loops.</p>
AE Program: <Program_Name>	This sub-section shows all the SQL Actions for a particular program. The Action properties that impact performance are flagged. For example, BulkInsert. ReUse is not flagged because it is self-evident when the Execute count is higher than the compile count.



Keep in mind that when you run a SQL trace at the Application Engine level and the PeopleTools level simultaneously you may see misleading results. In short, extra overhead gets added to the overall SQL timings by the PeopleTools trace. Tracing SQL at the Application Engine level (-TRACE) adds to the non-SQL times because PeopleTools writes the trace data after timing the SQL.

PeopleCode Actions (Section 2)

The second section, or PeopleCode section, records the performance associated with all the PeopleCode Actions in your program.

Column/Section	Description
PeopleCode	This column contains all the names of the PeopleCode Actions in your program.
Call	This column shows how many times each PeopleCode Action gets called during the program run.
Non-SQL	This column shows the time consumed by your PeopleCode Actions that does not involve any SQL.
SQL	This column shows the time consumed by your PeopleCode Actions executing SQL. The SQL time total should be similar to that of the PeopleCode SQL subsection the first section of the trace file.
Total	The total indicates the cumulative amount of time spent in the Action.

The following example shows a sample PeopleCode Actions section.

	Call	Non-SQL	SQL	Total
PeopleCode	Count	Time	Time	Time

AE Program: BIIF0001				
AE.UPD_AE	16	0.0	0.0	0.0
ASGNIVC1.NTMP	1	0.0	0.0	0.0
ASGNIVC1.REGISTER	1	0.0	0.0	0.0
<snip>				
NOTE.NEXTSEQ	7	0.0	0.0	0.0
OPTEDIT.UPD2NEW	1	0.0	0.0	0.0
UPDDSLVL.INIT	9	0.0	0.0	0.0
UPIVCAMT.UPCHDRAM	7	0.0	0.0	0.0
		-----	-----	-----
		2.5	0.2	2.7



Note. The columns containing all zeros but totaling 2.5, for example, is somewhat misleading. Keep in mind that the system rounds to the first decimal place (tenths) but only after calculating the sum of each Action time. You can assume that for the previous example, each Action required .09 seconds or less to complete.

PeopleCode Built-ins and Methods (Section 3)

The third section contains either a list of all of the PeopleCode built-ins and methods used or a summary thereof. To get a list of all of the built-ins and methods, you need to enable the PeopleCode Detail Timings in addition to the Statement Timings trace.

If a method or built-in function consumes a large amount of time, you may want to consider alternatives. For example, if array processing dominates your run time, consider inserting the data into temporary tables and performing the processing on tables in the database.

The following example shows a trace with the PeopleCode Detail Timings trace turned on.

E x e c u t e

PEOPLECODE Builtin/Method	Count	Time
-----	-----	-----
Decimal(Type 0) Builtin Len	9	0.0
Decimal(Type 0) Builtin RoundCurrency	52	0.0
String(Type 1) Builtin LTrim	16	0.0
<snip>		
Array(Type 263) Builtin CreateArrayRept	6	0.0
Array(Type 275) Method Push	6	0.0

The following example shows a trace with the PeopleCode Detail Timings trace turned off.

E x e c u t e		
PEOPLECODE Builtin/Method	Count	Time
-----	-----	-----
Any(Type 4) BuiltIns	8	0.0
SQL(Type 130) Methods	8	0.1
SQL(Type 130) BuiltIns	2	0.0
Record(Type 131) BuiltIns	1	0.2

Summary Data (Section 4)

The fourth section contains summary data. The values in this section reveal an overview of the program run without drilling down too far into the details.

Total run time	:	50.8		
Total time in application SQL :	42.1	Percent time in application SQL :		
83.0%				
Total time in PeopleCode	:	2.5	Percent time in PeopleCode	:
5.0%				
Total time in cache	:	2.5	Number of calls to cache	:
257				

The following table describes the values that appear in this section.

Value	Description
Total run time	This value presents the overall amount of time a program required to complete from start to finish.
<i>SQL</i>	
Time in application SQL	This value represents all of the time that your program spent executing SQL. The value includes SQL executed by both PeopleCode and SQL Actions.
Percent time in application SQL	This value represents the percentage of time spent executing SQL compared to the entire program run.
<i>PeopleCode</i>	
Time in PeopleCode	This value represents all of the time that your program spent executing PeopleCode. Time in PeopleCode excludes SQL executed from within PeopleCode.
Percent time in PeopleCode	This value represents the percentage of time spent executing PeopleCode compared to the entire program run.
<i>Cache</i>	
Total time in Cache	This value represents the amount of time your program spent retrieving objects from cache or refreshing cache. Time in cache includes all memory cache access, file cache access and SQL executed to load managed objects, such as Application Engine program components, meta-data, and so on. Time will vary according to where Application Engine finds an object. For instance, retrieving an object that the system cached during a previous run will be faster than retrieving it from the database.
Number of calls to Cache	This value represents the actual number of calls your program made to cache. The number of calls to cache will remain constant for the same Application Engine program processing the same data.

Environment Information (Section 5)

The fifth section contains environment information specific to Application Engine. If your programs appear to be performing poorly, you should always check the trace value that you have set.

Each trace produces an unavoidable degree of overhead. As such, the more traces you have enabled the more likely you are to see degraded performance. Make sure that you are only running the traces you need. This section shows you the following:

- PeopleTools SQL Trace
- PeopleTools PeopleCode Trace
- Application Engine Trace

- Application Engine DbFlags (%UpdateStats)

Table

You also have the option of writing the Statement Timings traces to a table. By storing timings information in a table, you can store historical data in the database, which allows you to produce reports that aid in trend analysis, allow ad hoc SQL queries for longest running statements, and so on. With the timings data stored in the database, you can manipulate and customize reports to only show the metrics in which you are most interested.

You can use third party tools to query and present the data in such ways as detailed graphical representations of your program's performance. You can also implement alarms if the performance of a program reaches a specified maximum value in a particular area such as SQL Compile time.

The Statement Timings trace populates the following tables. For convenience, we've included the columns and schema for the tables associated with this trace.

PS_BAT_TIMINGS_LOG (Parent)

This table stores the more general information for a program run.

BAT_TIMINGS_LOG (Record)				
Record Fields		Record Type		
	Num	Field Name	Type	Short Name
	1	PROCESS_INSTANCE	Nbr	Instance
	2	PROCESS_NAME	Char	Proc Name
	3	OPRID	Char	User
	4	RUN_CNTL_ID	Char	Run Cntl
	5	BEGINDTM	DtTm	Begin Date/Time
	6	ENDDTTM	DtTm	End Date/Time
	7	TIME_ELAPSED	Nbr	Elapsed
	8	TIME_IN_PC	Nbr	In PeopleCode
	9	TIME_IN_SQL	Nbr	In SQL
	10	TRACE_LEVEL	Nbr	AE Trace level
	11	TRACE_LEVEL_SAM	Nbr	SAM Trace Level

PS_BAT_TIMINGS_DTL (Child)

This table stores the more granular details associated with a program run, such as Execute Count, Fetch Time, and so on.

BAT_TIMINGS_DTL (Record)						
Record Fields		Record Type				
Num	Field Name	Type	Len	Format	Short Name	Long Name
1	PROCESS_INSTANCE	Nbr	10		Instance	Process Instance
2	BAT_PROCESS_TYPE	Char	1	Upper	Process type	Process type (cobol, AE,
3	BAT_PROGRAM_NAME	Char	12	Upper	Program	Program or module name
4	BAT_DTL_TYPE	Char	1	Upper	Detail type	Detail type (SQL, People
5	DETAIL_ID	Char	100	Mixed	Detail ID	Detail line identifier
6	COMPILE_COUNT	Nbr	10	Raw B	Compile Count	Count
7	COMPILE_TIME	Nbr	15		Compile Time	Time (in milliseconds)
8	EXECUTE_COUNT	Nbr	10	Raw B	Count	Count
9	EXECUTE_TIME	Nbr	15		Time	Time (in milliseconds)
10	FETCH_COUNT	Nbr	10	Raw B	Fetch Count	SQL Fetch Count
11	FETCH_TIME	Nbr	15		Fetch Time	SQL Fetch Time
12	RETRIEVE_COUNT	Nbr	10	Raw B	Count	Count
13	RETRIEVE_TIME	Nbr	15		Time	Time (in milliseconds)
14	BULK_INSERT	Char	1	Upper	Bulk Insert fla	Bulk Insert flag
15	PEOPLECODECOUNT	Nbr	10		PC Count	PeopleCodeCount
16	PEOPLECODESQLTIME	Nbr	15		Time	PeopleCodeSQLTime
17	PEOPLECODETIME	Nbr	15		PC Time	PeopleCodeTime
18	CURRENCY_ROUND	Char	1	Upper	Currency roundi	Currency rounding flag
19	EXECUTE_EDITS	Char	1	Upper	Execute edits f	Execute edits flag

PS_BAT_TIMINGS_FN

This table stores the PeopleCode Detail Timings information.

BAT_TIMINGS_FN (Record)						
Record Fields		Record Type				
Num	Field Name	Type	Len	Format	Short Name	Long Name
1	PROCESS_INSTANCE	Nbr	10		Instance	Process Instance
2	DETAIL_ID	Char	100	Mixed	Detail ID	Detail line identifier
3	EXECUTE_COUNT	Nbr	10	Raw B	Count	Count
4	EXECUTE_TIME	Nbr	15		Time	Time (in milliseconds)

BATTIMES.SQR

PeopleSoft provides BATTIMES.SQR as an example of the type of reports you can generate to reflect the information stored in the previous BAT_TIMINGS tables.

You can produce a summary report for all the programs for a specific Run Control ID that looks similar to the following:

BATTIMES_30.spf - SQR Viewer

File Edit View Page Help

PeopleTools 8.1
Copyright (c) 1988-2000 PeopleSoft, Inc.
All Rights Reserved
All timings in seconds

PeopleSoft Stored Batch Timings Summary Report

Process Name	: AETESTPR0G	Process Instance	: 2
Run Control ID	: greg	Operator ID	: PTDM0
Total run time	: 5.5	Run Start	: 04/24/2000 10:42:34
		Run Complete	: 04/24/2000 10:42:40
Total time in application SQL	: 1.9	Percent time in application SQL	: 33.7
Total time in PeopleCode	: 0.3	Percent time in PeopleCode	: 4.7
AE Trace Setting	: 1024	PeopleTools SQL Trace Setting	: 31

Process Name	: AETESTPR0G	Process Instance	: 14
Run Control ID	: greg	Operator ID	: PTDM0
Total run time	: 3.1	Run Start	: 04/24/2000 11:03:16
		Run Complete	: 04/24/2000 11:03:19
Total time in application SQL	: 0.8	Percent time in application SQL	: 26.2
Total time in PeopleCode	: 0.0	Percent time in PeopleCode	: 1.0
AE Trace Setting	: 1024	PeopleTools SQL Trace Setting	: 31

Process Name	: AETESTPR0G	Process Instance	: 15
Run Control ID	: greg	Operator ID	: PTDM0
Total run time	: 3.5	Run Start	: 04/24/2000 11:03:27
		Run Complete	: 04/24/2000 11:03:30
Total time in application SQL	: 0.8	Percent time in application SQL	: 22.2
Total time in PeopleCode	: 0.3	Percent time in PeopleCode	: 9.0
AE Trace Setting	: 1024	PeopleTools SQL Trace Setting	: 31

For Help, press F1

NUM 1/1

Summary Report for all program runs from a Run Control

Or, you can get detail data for a specific Process instance that looks similar to the following:

BATTIMES_29.spf - SQR Viewer

File Edit View Page Help

PeopleTools 8.1
Copyright (c) 1988-2000 PeopleSoft, Inc.
All Rights Reserved
All timings in seconds

PeopleSoft Stored Batch Timings Detail Report

Process Name	: AETESTPR0G	Process Instance	: 28
Run Control ID	: greg	Operator ID	: PTDM0
Total run time	: 4.3	Run Start	: 04/24/2000 12:10:37
		Run Complete	: 04/24/2000 12:10:41
Total time in application SQL	: 0.9	Percent time in application SQL	: 20.7
Total time in PeopleCode	: 0.1	Percent time in PeopleCode	: 1.4
AE Trace Setting	: 1024	PeopleTools SQL Trace Setting	: 31

SQL Statement	C o m p i l e		E x e c u t e		F e t c h		Total Time
	Count	Time	Count	Time	Count	Time	
AE Internal							
COMMIT	0	0.0	15	0.1	0	0.0	0.1
LOGMSG	0	0.0	3	0.3	0	0.0	0.3
							0.4
AETESTPR0G							
DBSTATS.UPDSTATS.S	1	0.0	1	0.2	0	0.0	0.2
MAIN.STATS.D	1	0.0	1	0.0	3	0.0	0.0
STATS.SECTIONS.S	1	0.0	2	0.0	2	0.0	0.1
STATS.STEPS.S	2	0.0	2	0.0	2	0.0	0.0
STATS.STMTS.S	2	0.0	2	0.0	2	0.0	0.0
TESTNUM.INTEGER.S	1	0.0	1	0.0	1	0.0	0.0
TESTNUM.INTRESLT.S	1	0.0	1	0.0	1	0.0	0.0

For Help, press F1

NUM 1/1


Detail Timings Report

You invoke the BATTIMES.SQR through the Process Scheduler using the following procedure.

To invoke Batch Timings

1. Open Process Scheduler Manager.
2. Click Use, Batch Timings, and click either Add or Update/Display.

The Batch Timings page appears.

3. From the **Report Type** dropdown list, select *Detail* or *Summary*.
4. In the **Batch Timings For** group box, enter the Run Control ID for Summary reports, and enter the specific Process Instance for Detail reports.
5. When you have made the appropriate selections, click the  button.

To view the Batch Timings using Process Monitor

6. Open Process Monitor.
7. Locate the program run associated with the current trace.
8. Click the **Details** button.
9. On the **Process Detail** dialog box, click the **Batch Timings** link.

On the Batch Timings dialog box, the PeopleCode Detail Timings do not appear. They only appear in the file format.

DB Optimizer Trace

With the DB Optimizer trace you can write the trace to a file or a table. The DB Optimizer trace reveals the execution/query plan for the SQL that your Application Engine program generates. Each SQL statement only gets traced once.

How you view the results of this trace depends on the type of RDBMS you're currently using. For instance, on some platforms only the trace to file option is available, whereas on others only the trace to table option is available. The following table shows the options available for each of the platforms PeopleSoft supports.

RDBMS	File	Table
Microsoft SQL Server	X	X
DB2 for OS/390		X
DB2 for UDB (AIX, Solaris, NT)		X
Oracle	X	X
Informix	X	

<i>RDBMS</i>	<i>File</i>	<i>Table</i>
Sybase	N/A	N/A



Note. This trace is not supported on Sybase.

The following information offers some details related to the implementation of this way to facilitate analysis of the trace results. Examine the information that pertains to your RDBMS.



PeopleTools does not collect optimizer data for SQL originating from PeopleCode Actions, except in the following circumstances. If you run Oracle and Informix and run an optimizer trace to file, the system traces *all* SQL that executes after the first SQL Action executes.

Microsoft SQL Server

On Microsoft SQL Server both the file and the table options are available for the DB Optimizer trace.

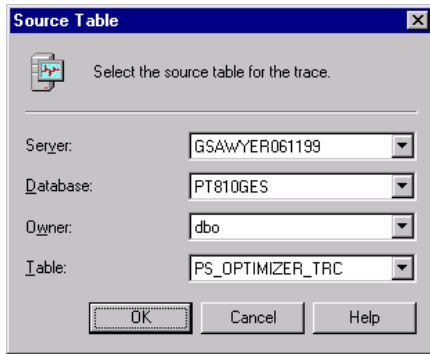
For the file option, Application Engine writes the DB Optimizer traces to the following location: %TEMP%\psms<queueid><spid>.trc. To read the trace, you will need to use the SQL Server Profiler utility.



Note. The trace file will be written to the server directory when you've specified the trace on the client. If the client has %Temp% set to a drive or directory that does not exist on the server, Application Engine does not generate a trace file.

For the table option, Application Engine writes the trace data to the following table: dbo.PS_OPTIMIZER_TRC. PeopleTools creates the table automatically when you run the trace for the first time. The trace data written to the table is identical to the data appearing in the optimizer trace file.

You also need to use the SQL Server Profiler utility to view the optimizer results. To view the populated trace table you need to specify the current server and database on the Source Table dialog, and the Owner and Table values need to be *dbo* and *PS_OPTIMIZER_TRC* respectively. For example,



Source



For more information on the SQL Server Profiler utility refer to your Microsoft documentation.

In the trace, you will find information regarding Text, Duration and StartTime for the following:

- Execution Plans
- Remote Procedure Calls
- INSERTs (UPDATEs, DELETEs, and SELECTs)
- PeopleSoft generated user events associating trace data with a PeopleSoft SQL identifier

If the Application Engine program aborts while you're using this trace option, you must check that Application Engine was not tracing a SQL statement at the moment the program aborted. If it was tracing a SQL statement at that time, you must manually kill the trace. Otherwise the trace thread on the server will continue to run and lock the trace file, and each time that SPID gets reused by the server, new information will be appended to the locked trace file.

To stop the trace manually, submit the following command from the Query Analyzer:

```
xp_trace_destroyqueue <queueid>
```

The <queueid> in the file name %TEMP%\psms<queueid><spid>.trc is the ID corresponding to the queue used for the first SQL statement that the system profiled. Since this trace is only designed to trace Application Engine SQL (not PeopleTools SQL), we close the queue after every statement profiled. Therefore the queue that needs to be destroyed may not be the Queue ID used in the trace file.



If the %TEMP% variable is set to a location that does not exist, Application Engine does not generate a trace file.



For more information refer to the PeopleSoft Installation and Administration guide for Microsoft SQL server

Oracle

With the trace file option, Application Engine writes the trace file to the default Oracle trace directory specified on the database server. To read the trace file, use the TKPROF utility.

If you want to use the DB Optimizer(Table) option on Oracle, a PLAN_TABLE must exist and the statement_id must be varchar2(254) instead of varchar2(30).



For more information on creating a PLAN_TABLE and the complete PLAN_TABLE schema refer to your Oracle documentation.

With the table option, PeopleSoft updates the trace rows as follows:

- **EXPLAIN PLAN SET STATEMENT_ID.** PeopleSoft updates the STATEMENT ID column:

```
EXPLAIN PLAN SET STATEMENT_ID = ApplId.Section.Step.Type FOR <sqlstmt>
```

- **PLAN_TABLE's REMARKS column.** PeopleSoft updates the REMARKS column:

```
PLAN_TABLE's REMARKS column = 'ProcessInstance-RunControlId(QueryNo)'
```

Where QueryNo is a count of how many SQL statements have been traced up to a particular point.



When tracing to a table with Oracle, PeopleSoft does not perform optimizer traces on %UpdateStats and %TruncateTable unless the latter resolves into a DELETE. On the other hand, Oracle's TKPROF to file, handles both the ANALYZE and TRUNCATE commands.

Informix

For Informix, Peoplesoft only supports the trace file option. Where the trace file gets written depends on the operating system on which your database server runs.

- **UNIX.** For UNIX, Application Engine writes the plan to the sqexplain.out file. If the client program runs on the same machine as and the database server, the sqexplain.out file appears in the current directory. When the current database is on another computer, the sqexplain.out file gets written to the PeopleSoft owner's directory on the remote host.
- **Windows NT.** For Windows NT, Application Engine writes the plan to the following file: INFORMIXDIR%\sqexpln\username.out.

DB2 for OS/390

For DB2 for OS/390, PeopleSoft only supports the table option. PeopleSoft has implemented the following to facilitate this trace:

- PeopleSoft selects the max QueryNo from the PLAN_TABLE, increments it by 1000 to avoid clashing with other processes and then increments it by 1 for every SQL statement traced.
- PeopleSoft sets the following parameter: SET REMARKS = ApplId.Section.Step.Type-RunControllId(ProcessInstance)

DB2/UDB

For DB2 for UNIX, PeopleSoft only supports the table option. To facilitate this trace for DB2/UNIX, PeopleSoft has implemented EXPLAIN ALL SET QUERYNO =ProcessInstance SET QUERYTAG = 'Section.Step' FOR <sql stmt>.

Disabling the DB Optimizer Trace

While the DB Optimizer Trace is enabled, performance may be affected. Typically, you turn on this trace only during periods in which you are collecting detailed performance metrics. When you are not tuning your performance, the DB Optimizer trace should be turned off.

To prevent an administrator, or perhaps a user, from unwittingly turning the optimizer trace on or leaving it on after doing performance tuning, you can disable the DB Optimizer trace for an entire database.

For example, say you have a production and a development database, you might want to enable the optimizer trace for the development database while disabling the optimizer trace for the production database.

On the PeopleTools Options page, the **Allow DB Optimizer Trace** option enables you to control whether or not a user can initiate the Application Engine DB Optimizer Trace on the database.

If **Allow DB Optimizer Trace** is selected, then the DB Optimizer Trace is a valid tracing option on the database. On the other hand, if **Allow DB Optimizer Trace** is deselected then when any user attempts to initiate the optimizer trace, the following error message appears:

```
PeopleTools 8.10 - Application Engine
```

```
Copyright (c) 1988-2000 PeopleSoft, Inc.
```

```
All Rights Reserved
```

```
Optimizer trace request ignored, because disallowed in PSOPTIONS.
```

Temporary Table Performance Considerations

Before implementing dedicated temporary tables for parallel processing, there are some performance considerations.



For more information on implementing temporary tables, see [Using Temporary Tables](#).

Initial Estimates

Keep in mind that if you find that you need more temporary table instances after you've entered production, you need to rebuild all of your temporary tables so that the database reflects the proper inventory of instances. While the build process runs, users can't access the database. Because of this, it is important to spend time deriving adequate estimates as to the number of temporary tables required.

Online Temporary Table Allocation

Within the database there exists a physical table designed to store online temporary table instance usage. The table is named PS_AEONLINEINST.

If you notice performance issues related to online Application Engine program runs, enable the Application Engine SQL and Timings trace. If the following SQL command

```
INSERT INTO PS_AEONLINEINST ...
```

appears to be requiring too much time to complete, this is a good indication that there aren't enough online temporary instances defined on PeopleTools Options.

Viewing Temporary Table Usage

If you have implemented temporary tables for parallel Application Engine program runs, you use the Temporary Table Usage page to find out how the system allocates temporary tables to your programs. To access this page select **PeopleTools, Application Engine, Inquire, Temporary Table Usage**.

Parallel processing is designed to be a performance enhancing option. However, if the demand for temporary table instances consistently exceeds the current supply, your performance suffers. Also, in other situations your inventory of temporary table instances may far outnumber demand. Here, you may consider reducing the number of instances provided to conserve system resources.

Temporary Table Usage

View Temporary Instance Use By

Record (Table) Name:
Program Name:
 Refresh

Record (Table) Name	Total Instances	Locked Instances	Unused Instances	Select
AEXT_TAO	5	0	5	Select

Application Engine, Inquire, Temporary Table Usage

In the View Temporary Instance Use By section you have two controls that enable you to specify search criteria. You can view usage by the Record (Table) Name or you can view by Program Name. You use the lookup buttons to select a particular record or program. To refresh the contents of the page, click the Refresh button.

This page shows you the following metrics for evaluating your inventory and allocation of temporary tables:

- **Program Use Count.** Reveals the total number of programs that use a particular temporary table.
- **Total Instances.** Shows the total number of instances of a temporary table that exist.
- **Locked Instances.** This value indicates the current number of instances that they system has locked for program runs.
- **Unused Instances.** This value indicates the current number of instances that are available for use.

Managing Abends

If you are taking advantage dedicated temporary tables for Application Engine programs, then you need to know how to free, or unlock, a temporary table in the event that the program running against it abends. Because most Application Engine programs run through Process Scheduler, typically you just use the Process Monitor to unlock the temporary tables. Deleting or restarting a process using the Process Monitor automatically frees the locked temporary tables.



For more information on the Process Monitor, see Process Scheduler.

For the programs that you invoke outside of Process Scheduler, PeopleTools provides the Manage Abends page. Programs running outside of Process Scheduler include those invoked from CallAppEngine PeopleCode and the command line.

Temp Tables	Process Instance	Run Control ID	Program Name	User ID	Run Date and Time
Temp Tables	25	DOCTEST	BULKTEST	PTDMO	05/17/2000 1:09:57PM
Temp Tables	29	DOCTEST	AETESTPROG	PTDMO	05/17/2000 1:56:27PM

Manage Abends Interface

Use the following procedure to free temporary tables locked by a program abend.

To free locked temporary tables using the Manage Abends page

1. Navigate to the Manage Abends page by selecting **Application Engine, Use, Manage Abends**.
2. Identify the program that has the particular temporary table(s) locked.

You can uniquely identify programs using the Process Instance, Run Control ID, Program Name, User ID, and Run Date and Time columns.
3. Click the Temp Tables link.
4. On the Temporary Tables sub-page, click the Release button to unlock the temporary tables associated with the program.

Restarting Application Engine Programs

One key feature of Application Engine is its built-in checkpoint/restart capabilities. With Application Engine, developers don't need to code restarts on their own. Application Engine, by default, uses each Step that executes successfully as a checkpoint. So, if there is a failure on the next step, the end user can restart the request from the last successful checkpoint, or the Step immediately preceding the step that failed. All of this is transparent to the end user who merely needs to restart the program from the Process Request page.

After reading this section, you'll become familiar with the methods you can use to restart an Application Engine program. End users that typically invoke Application Engine programs should also be informed of the appropriate methods used to restart Application Engine programs.

Restarting an Application Engine Program

There are two methods for restarting an Application Engine program. You can use the command line, or you can use the Process Requests page to restart your Application Engine program. As you'll see in the following sections, which option you use to restart an Application Engine program depends on who's restarting the program run and the run location of the program.



Note. The following procedures for restarting a failed Application Engine program assume that you have rectified the error that caused the program to fail in the first place. For instance, suppose the name of a referenced table has changed. Regardless of how many times you restart the program, it will continue to fail until you've modified references to the old table name.

Command Line

Normally, using the command line for restarting your Application Engine programs is reserved for developers and system administrators. End users, in most cases, should not be expected to use this method to restart programs that fail.

You can use the command line option to restart programs that have a run location of either Client or Server. Application Engine only references the Process Instance of the failed process.

So, with that being the case, if you ran a process on the client and it failed, you can restart it from the server using the server command line. Likewise, if you ran a process from the server and it failed, you can restart it from the client using the command line.

To restart an Application Engine program from the command line

1. Collect the command line values associated with the failed program.

These values include, database type, database name, Operator ID and password, Run Control ID, program name, and the Process Instance. You can find these variables on the Process Details dialog, the corresponding State Record, or the Application Engine Run Control table. Where the values reside depends on how you invoked the program. For instance, if you invoked the program using the command line, or outside of Process Scheduler, then you will not be able to view any details associated with the program run on the Process Details dialog.

2. Enter the following command line syntax at the command prompt substituting the values from the previous step.

```
PSAE.EXE -CT <DB_TYPE> -CD <DB_NAME> -CO <OPRID> -CP <PASSWORD> -R <RUN_CONTROL>  
-AI <PROGRAM_NAME> -I <PROCESS_INSTANCE>
```



Note. Some RDBMS platforms, such as Sybase, also require that you include a server name in the argument list.



For more information on the command line variables, see [Invoking Application Engine Programs](#).


Process Request Page

With the Process Requests page, you only can restart Process Requests that have a Run Location of Server. Consequently, to restart an Application Engine program that ran on the client, it's necessary to use the command line method. Most end users will require assistance with the command line interface. For that reason, it's a good idea to limit the number of programs that run on the client.

To restart an Application Engine program from the Process Request page

1. Open Process Scheduler by selecting Go, PeopleTools, Process Scheduler Manager.
2. Select Inquire, Process Requests.
3. Locate the Instance Number and row associated with the program that you need to restart.

The Run Status column should read "No Success".

4. To reveal the information regarding the failed process, click the Process Detail button  on the right-hand side of the interface.
5. On the Process Request Details dialog, in the Update Process group, select Restart Request, and click OK.

Deciding to Start over or Restart

When an Application Engine program ends abnormally, the question arises regarding whether you should restart the process or just start it from the beginning. Keep in mind that your Application Engine program ran at least part way through, so starting over may leave your data in a "weird" state. Also, there may be some application logic that would need to be "undone" depending on what stage the program was at when it failed, what the program had committed, and so on.

Ultimately, the answer to the question hinges on how you coded your Application Engine program. Can the logic accommodate a restart? If the answer is *No*, then you should probably consider starting the program over from the beginning.



Note. By selecting the Disable Restart check box on the Program Properties dialog you can specify the process to start each time as if it were "new."



For more information on developing for restart, see [Developing for Restart](#).

If you decide that starting your program over from the beginning is the best alternative, keep the following in mind. If you try to start, from the beginning, a new process with the same Run Control ID and OPRID as the process in a "restartable" or "No Success" state, you'll receive a

"suspend" error. Consequently, to start the program over from the beginning, you will need to use SQL to delete the row that corresponds to the failed program from the Application Engine Run Control table and your State Record.

To start an Application Engine program from the beginning (again)

1. Open your native SQL editor, and manually delete the row in PS_AE_RUN_CONTROL that corresponds to the program you want to start from the beginning.

Use the following SQL to accomplish this step:

```
DELETE FROM PS_AE_RUN_CONTROL  
  
WHERE OPRID=<OPRID>  
  
AND RUN_CNTL_ID=<Run_Control_ID>
```

2. Delete from your State Record the row that corresponds to the failed program run.

Use the following SQL to accomplish this step:

```
DELETE FROM PS_<MY_AET>  
  
WHERE PROCESS_INSTANCE=<Process_Instance>
```

Troubleshooting

The following topics offer some brief troubleshooting tips for the most common errors associated with restarting Application Engine programs.

"Bad restart" Messages

In order for a restart attempt to actually restart the process, the process can't be a completed process. If you attempt to restart what Application Engine believes to be a process that completed successfully, you'll get a "bad restart" message. You can also get this message if your Application Engine application is defined with restart disabled.

The "Suspend" Error

A suspend error occurs when you attempt to start a new process that matches the Run Control ID and OPRID for another process in a "restartable" state. Of course, the Process Instance for these two processes is different. In this scenario, the new request will fail.

This usually occurs when an end user tries to run the program again after receiving an error on the previous attempt.

To resolve this issue you have the following options:

- Clear the information associated with the process from the Run Control table and State Record,

and start the process over from the beginning.

- Just select Restart Request from the **Process Request Details** dialog.



Note. By "restartable," we mean that Disable Restart is not selected on the Program Properties dialog. Consequently, the program is restartable.

Index

%

- %AeProgram 4-42
- %AeSection 4-42
- %AeStep 4-43
- %AsOfDate 4-43
- %Bind 4-28
- %ClearCursor 4-39
- %Comma 4-43
- %Execute 4-40
- %ExecuteEdits 4-31
- %JobInstance 4-43
- %LeftParen 4-43
- %Next 4-41
- %Previous 4-41
- %ProcessInstance 4-43
- %ReturnCode 4-43
- %RightParen 4-43
- %RoundCurrency 4-41
- %RunControl 4-43
- %Select 4-32
- %SelectInit 4-33
- %Space 4-43
- %SQL 4-34
- %SQLRows 4-43
- %Table 4-19, 4-35
- %TruncateTable 4-20, 4-36
- %UpdateStats 4-36

A

- abends 5-37
 - managing 5-37
- Abends 4-13
- Abort 3-34
- Action Types 3-25
- Actions 3-24
 - Call Section 3-34
 - changing type 2-23
 - DO Select 3-31
 - DO Until 3-33
 - Do When 3-31
 - Do While 3-31
 - execution hierarchy 3-38
 - inserting 3-26
 - Log Message 3-37
 - no rows 3-29
 - PeopleCode 3-33, 4-49
 - program flow 3-30

- properties 3-24
- ReUse 3-28
- SQL 3-27
- Administration
 - Application Engine 5-1
 - troubleshooting 5-41
 - restarting Application Engine programs 5-38, 5-40
 - command line 5-39
 - process request page 5-40
- Advanced
 - properties 3-7
- AESection Object 4-51
- API
 - PeopleTools 4-59
- Application Designer
 - Application Engine 2-1
- Application Engine
 - abends 5-37
 - action types 3-25
 - Actions 3-24. *See* Actions
 - actions, defined 1-8
 - administration 5-1
 - Advanced properties 3-7
 - advanced topics 4-1
 - advantages 1-2
 - user interface 1-4
- Application Library 3-8
- commit *See* Commit
- CommitWork 4-59
- creating programs 3-1
- debugging 4-78
- debugging commands 4-81
- debugging options 4-81
- defined i, 1-1
- designer interface 2-1
 - menus 2-8
- navigation 2-6
- overview 2-1

- dynamic SQL 4-77
- enabling debugging 4-78
- encapsulation 1-3
- execution precedence 3-16
- file layout object 4-54
- general properties 3-3
- IF, THEN logic 4-47

- inserting actions 3-26
- introduction 1-1
- invoking programs 5-2
- locating sections 3-18
- Macros 4-39
- meta-SQL 1-5
- Meta-SQL 4-26
- no rows 3-29
- object workspace 2-3
- overview 1-1
- PeopleCode
 - Action execution 4-49
 - in loops 4-50
 - math functions 4-61
 - SQL Objects 4-62
 - variable scope 4-48
- PeopleCode actions 1-9
- PeopleCode notes 4-60
- platform flexibility 1-6
- Process Scheduler 5-3
- program flow 3-30
- program flow actions 1-8
- program properties 3-2
- programs, defined 1-8
- project workspace 2-2
- Remote Call 4-55
- restart 1-4. *See* Restart
- restarting programs 5-38
- reusing SQL 4-6
- reusing statements 3-28
- section object 4-51
- Section Properties 3-13
- sections, defined 1-8
- set processing 4-63
- sharing state records 3-36
- SQL actions 1-9, 3-27
- State Records 1-10, 3-3
- statement timings trace 5-20
- steps, defined 1-8
- supported platforms 5-3
- switching views 2-6
- system variables *See* System Variables, Application Engine
- terminology 1-7
- tips 2-16
- TOOLSBSINRV 5-3
- traces 5-14
- tracing 5-13
 - enabling 5-14
- Tuxedo requirements 5-3
- upgrade 1-3
- using PeopleCode 4-44
- verses COBOL and SQR 1-2

- viewing definitions 2-3, 2-16
- viewing program flow 2-5
- Application Engine programs
 - copying 3-11
 - deleting 3-12
 - inserting sections 3-16
 - inserting steps 3-22
 - online calls to 4-51
 - opening 3-10
 - properties 3-10
 - renaming 3-11
 - set processing examples 4-70
 - step properties 3-20
 - synchronous calls to 4-51
- Application Library 3-8

B

- batch server
 - Application Engine considerations 5-3
 - TOOLSBSINRV parameter 5-3
 - Tuxedo requirements 5-3
- Batch Timings 5-20
 - table 5-28
- BATTIMES.SQR 5-29
 - invoking 5-31
 - viewing results 5-31
- Break 3-34
- building
 - temporary tables 4-17
- Bulk Insert 3-29, 4-7

C

- caching
 - Application Engine 5-12
- Caching
 - server 5-12
- Call Section
 - dynamic 3-35
 - program ID 3-35
 - program properties 3-36
 - section name 3-35
- Call Section Actions 3-34
- CallAppEngine 4-51, 5-8
 - defining global variables 4-46
 - events 4-52
 - fieldchange event 4-54
 - passing parameters 4-46
 - process instance 4-53
 - save event 4-53
- CD-ROM
 - ordering ii
- command line
 - tracing Application Engine programs 5-14
- Command Line

- Application Engine 5-9
- Commit 3-21, 4-4
 - Application Engine 3-21
 - frequency 3-21
- CommitWork 4-59
- Configuration Manager
 - tracing options for Application Engine 5-16
- creating Application Engine programs 3-8

D

- DB Optimizer trace 5-31
 - DB2 UDB 5-35
 - Disabling 5-35
 - Informix 5-34
 - Microsoft SQL Server 5-32
 - Oracle 5-34
 - OS/390 5-35
- DBFLAGS 5-11
- Debugging
 - Application Engine 4-78
- defining
 - temporary tables 4-17
- Definition Object
 - deleting 2-23
- Definition Objects
 - changing an Action type 2-23
 - cloning 2-24
 - copying 2-22
 - deleting 2-23
 - moving 2-23
 - renaming 2-23
 - working with 2-22
- Deleting
 - Application Engine programs 3-12
- Design time 2-1
- Development
 - advanced Application Engine topics 4-1
- Disable Restart 3-8
- disabling restart 4-9
- Do Select 3-32
- DO Select 3-31
 - considerations 3-32
- DO Until 3-33
- Do When 4-60
- DO When 3-31
- DO While 3-31
- Drop-Down Lists 2-20
- Dropdown Menus 2-10
- Dynamic SQL 4-60, 4-77

E

- Edit Menu 2-11, 2-12
- Execution
 - Application Engine Actions 3-38

F

- File Layout Object 4-54
- File Menu 2-10
- filtering sections 2-17
- Filtering Sections 3-15
- Find Object References
 - dialog box 3-19
- finding trace files
 - Application Engine 5-17
- folder icons 2-16

I

- Insert Menu 2-14
- invoking
 - Application Engine programs 5-2
- invoking Application Engine programs 5-3
 - command line 5-9

L

- locking
 - temporary tables 4-24
- Log Message Actions 3-37

M

- Macros
 - Application Engine 4-39
 - %ClearCursor 4-39
 - %Execute 4-40
 - %Next 4-41
 - %Previous 4-41
 - %RoundCurrency 4-41
- managing temporary tables 4-18
- Menus 2-8
- Message Set 3-8
 - Application Engine 3-8
- Meta-SQL
 - adjusting for temporary tables 4-19
 - Application Engine 4-26, 4-28
 - %Bind 4-28
 - %ExecuteEdits 4-31
 - %Select 4-32
 - %SelectInit 4-33
 - %SQL 4-34

%Table 4-35

%TruncateTable 4-36

%UpdateStats 4-36

referencing temporary tables 4-19

N

No Rows 3-29

O

Object

position 2-21

Object Layout 2-21

Object References

finding 3-19

Object Workspace 2-3

Objects

activating properties 2-19

selecting 2-21

opening Application Engine programs 3-10

Overview 2-1

P

pages

process requests 5-5

parallel processing 4-14

overview 4-14

temp tables 3-5

temporary tables 4-14

PeopleBooks

CD-ROM, ordering ii

printed, ordering ii

PeopleCode

AESction object 4-51

APIs 4-59

CallAppEngine 5-8

in Application Engine programs 4-44

loops 4-50

sequence numbering 4-61

State Records 4-47

PeopleCode actions

tracing 5-24

performance

temporary tables 5-36

Popup Menus 2-8

definition view 2-8

program view 2-9

Pop-Up Menus 2-7

process request

controls 5-6

pages 5-5

pages for Application Engine 5-5

upgrading pages 5-7

Process Scheduler

invoking Application Engine programs 5-3

Program Flow Actions 3-30

Application Engine 1-8

Program Flow View 2-7

Program ID 3-35

programs

abnormal ends 4-13

assigning temporary tables to 4-18

Programs

Application Engine 3-1

call section 3-34

creating 3-1

finding sections within 3-20

procedures 3-8

properties 3-2

sections 3-12

steps 3-20

Project Workspace 2-2

properties

general 3-3

state records 3-3

temp tables 3-5

Properties

Application Engine actions 3-24

edit boxes 2-19

PS_BAT_TIMINGS_DTL 5-15, 5-28

PS_BAT_TIMINGS_LOG 5-15, 5-28

PTPECOBL 4-55

R

RemoteCall 4-55

commit 4-58

PTPECOBL 4-55

restrictions 4-58

Renaming

Application Engine programs 3-11

Re-Select 3-32

Restart

deciding when to 4-10

developing for 4-8, 4-11, 4-12

disabling 3-8, 4-9

enabling 4-8

how it works 4-9

program level 4-11

section level 4-12

step level 4-12

Restartable 3-32

Restarting

Application Engine programs 5-38

restarting Application Engine programs

when to 5-40

ReUse 3-28

Rowsets 4-61

run time behavior
temporary tables 4-24

S

Section Filtering 2-17

Section Properties 3-13

Sections

- Application Engine 3-12
- calling dynamically 3-35
- Execution Precedence 3-16
- filtering 2-17, 3-15
- finding within current program 3-20
- inserting 3-16
- list of references 3-19
- locating 3-18
- navigating to 3-18

Select/Fetch 3-32

Set Processing 4-63

- advantages of 4-65
- defined 4-64
- examples 4-70
- planning 4-66
- platform issues 4-76
- requirements 4-66
- tips 4-69
- using 4-66

Sharing State Records 4-3

Skip Step 3-34

SQL

- bulk insert 4-7
- reusing 4-6
- set processing 4-63
- traces 5-19

SQL Actions

- ReUse 4-6

SQL counts 5-21

SQL Objects

- Application Engine 4-62

State Records 1-10, 3-3, 4-1

- overview 4-1
- PeopleCode 4-47
- record types 4-4
- sharing 3-36, 4-3
- using 4-1

statement timings 5-20

- file 5-21
- table 5-28

Statement Timings 5-20

Step

- status 3-22
- traces 5-18

Steps 3-20

- defining 3-20
- inserting 3-22
- properties 3-20

System Variables

Application Engine 4-42

- %AeProgram 4-42

- %AeSection 4-42

- %AeStep 4-43

- %AsOfDate 4-43

- %Comma 4-43

- %JobInstance 4-43

- %LeftParen 4-43

- %ProcessInstance 4-43

- %ReturnCode 4-43

- %RightParen 4-43

- %RunControl 4-43

- %Space 4-43

- %SQLRows 4-43

T

Temp Table Instances 4-22

temporary table

- Application Designer tasks 4-16
- creating 4-16
- instances 4-16

temporary table instances

- number of 4-22
- total 4-22

temporary tables

- adjusting meta-SQL for 4-19
- assigning 4-18
- batch mode 4-21
- batch processes 4-24
- building 4-17
- calling other programs 4-21
- clearing 4-20
- defining 4-17
- locking 4-24
- managing 4-18
- online mode 4-21
- online processes 4-23
- online vs. batch 4-23
- overview 4-14
- parallel processing 4-14
- PeopleTools Options 4-22
- performance considerations 5-36
- run time behavior 4-24
- sharing 4-21
- specifying number of 4-22

- using 4-14
- Temporary Tables 3-5
 - set processing 4-67
- Text Boxes 2-19
- Toolbar 2-15
- TOOLBINSRV 5-3
- Trace
 - locating Application Engine trace 5-17
- TraceAE 5-15, 5-16
- tracing
 - Application Engine program SQL 5-19
 - Application Engine program steps 5-18
 - Application Engine programs 5-13
 - Application Engine statement times 5-20
 - Application Engine traces 5-14
 - command line 5-14
 - configuration files 5-15
 - configuration options 5-16
 - enabling Application Engine traces 5-14
 - locating trace file 5-17
 - PeopleCode actions 5-24
 - PeopleCode built-ins 5-25
 - PeopleCode methods 5-25
 - understanding results 5-17
- Tracing
 - batch timings 5-20
 - command line (Application Engine) 5-14

- DB Optimizer 5-31
- triggering CallAppEngine 4-52
- Troubleshooting
 - Application Engine 5-41

U

- Upgrade Only 3-8
- upgrading request pages 5-7
- Using the Folder Icons 2-16
- Using the Plus and Minus Signs 2-16

V

- View Menu 2-7
- viewing definition objects 2-16
- Views
 - switching 2-7

W

- Workspace
 - refreshing 2-13