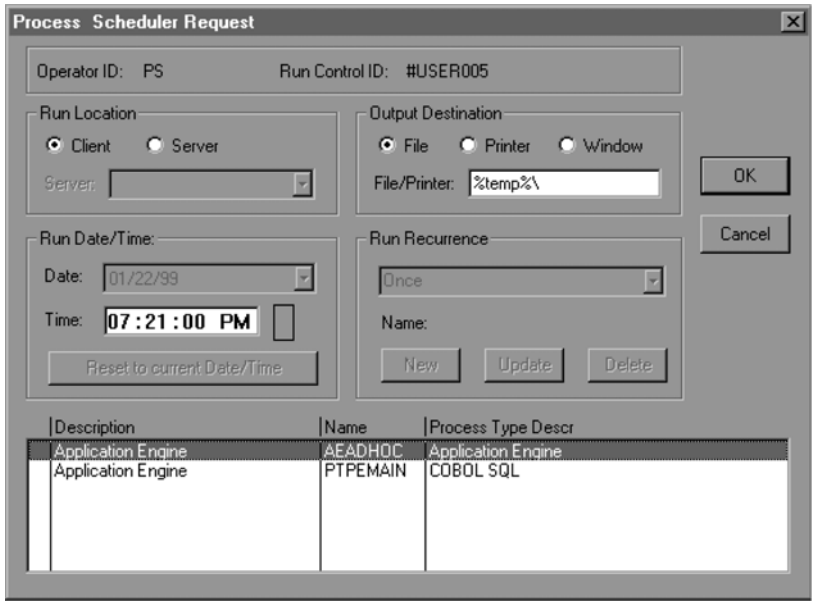


Figure 40.32 Submitting a Process Scheduler request

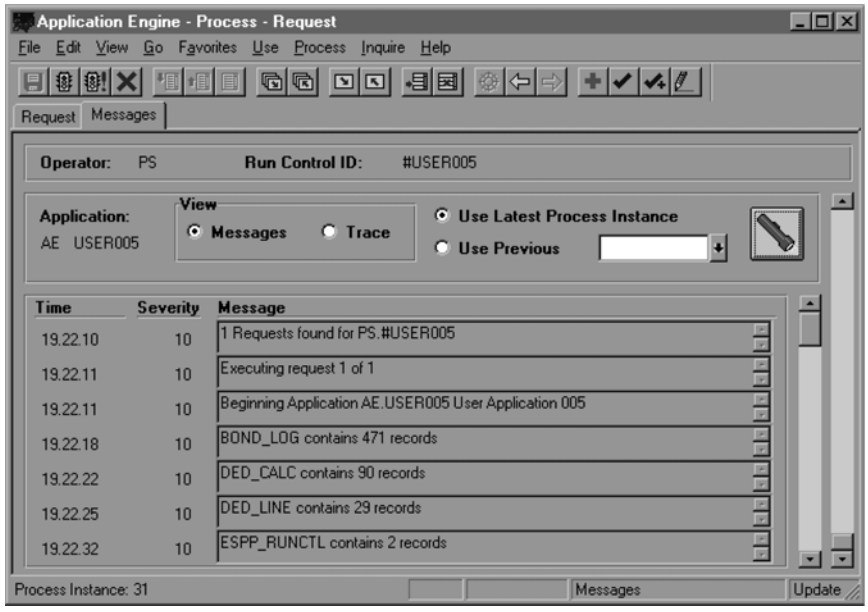


Figure 40.33 Reviewing Process Request messages

We are once again successful! The message log output matches that of our SQR version. Only tables that contain rows of data are displaying. Using the `DO When` construct, we've filtered out all tables with a zero row count. Two additional statement types—the `DO Until` and `DO While` statement types—control section execution in a similar fashion.

40.2 SQR/Application Engine comparison

Once again, let's take a look at the logical structure of both our programs:

SQR:

```
Begin-Program
        User prompted
        Main-Step1
                Count-Step1
                        Msg-Step1
```

Application Engine:

```
USER003
        Cache assignment
        MAIN.STEP1
                COUNT.STEP1
                COUNT.STEP2
                        MSG.STEP1
```

This time the structures are slightly different. The SQR program doesn't need an additional step to perform a decision. A simple `IF` statement is used to determine if the Msg-Step1 procedure should be performed. Application Engine requires the additional step to build a `DO When` condition. Based on the results, the MSG section is performed.

KEY POINTS

1. You can control the processing logic using a `DO When`, `DO While`, or `DO Until` statement. This adds decision-making capability to your program and can regulate which sections are performed.
2. The PSLOCK table is often used as a dummy table to evaluate `&BIND` data values. It can be used in the same manner as the DUAL table is used in Oracle.

CHAPTER 41

Dynamic sections

An Application Engine program has the ability to call a section dynamically. This is a very powerful feature. Sections may be created and, based on certain conditions, a particular section may be executed. Let's begin.

41.1 EXERCISE 6: CALLING DYNAMIC SECTIONS

Our exercise is simple. We are going to dynamically call a section that either writes the message "Hello World" or "Goodbye." Dynamic sections are called based on the contents of the cache field AE_SECTION. This field must exist in the cache record we've designated for our application. We'll populate this field on the Process Request panel with the name of the section we'd like to perform. We begin by displaying a simple SQR that prompts the user for their choice of messages to display. Keep in mind the SQR version isn't dynamic—it simply performs the routine based on user selection—but it will demonstrate the logic flow as if it were dynamic.

41.1.1 Creating an SQR version

```
! USER006.SQR

begin-program

input $choice 'Enter Section# (1=Hello 2=Goodbye)' maxlen=1

evaluate $choice
   when = '1'
      do Hello-Step1
   when = '2'
      do Goodbye-Step1
end-evaluate

end-program

begin-procedure Hello-Step1
show 'Hello World'
end-procedure

begin-procedure Goodbye-Step1
show 'Goodbye'
end-procedure
```

If the user enters a `'1'`, the SQR.log looks like this:

```
Hello World
```

If the user enters a `'2'`, the SQR.log looks like this:

```
Goodbye
```

Let's create a version of the program using Application Engine.

We begin by adding the USER006 application name and section MAIN (figure 41.1).

Navigation: Go →PeopleTools →Application Engine →Use →Application Engine → Application →Add

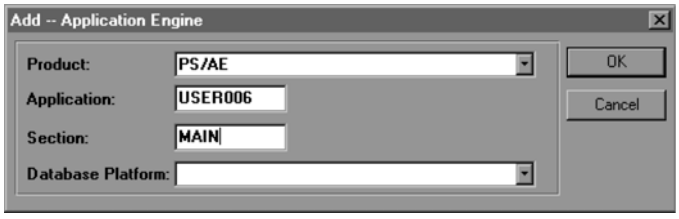


Figure 41.1
Naming the application

Fill in the description, version, and message set number as we've done in the past exercises. We're going to use a different cache record called AE_TESTAPPL_AET. This is a delivered PeopleSoft record. The field AE_SECTION is contained in this record so it's perfectly suited for our dynamic section exercise. We could have added this field to our USER_AET cache record, but I wanted to demonstrate the cache record assignment. There is no need to create additional cache records if one exists that meets your requirements.

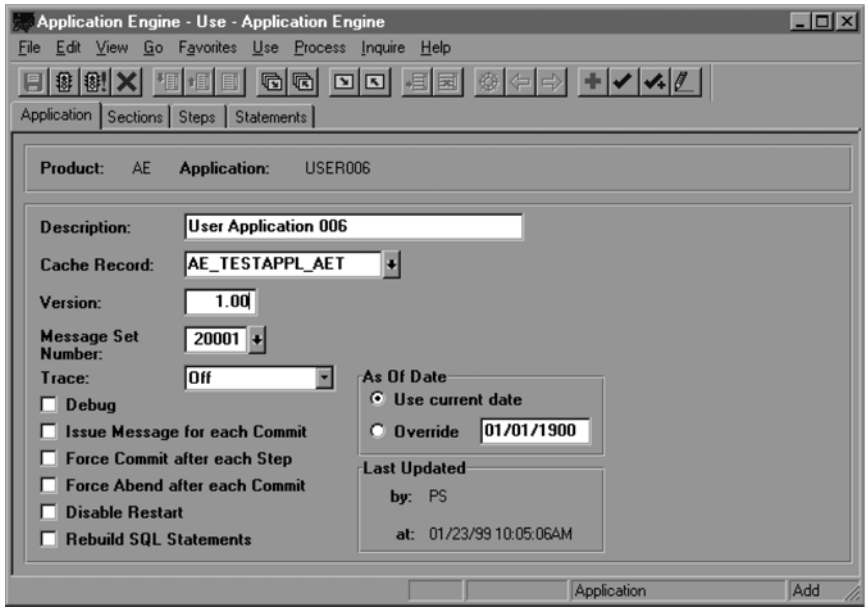


Figure 41.2 Defining the application

We add our section MAIN description (figure 41.3).

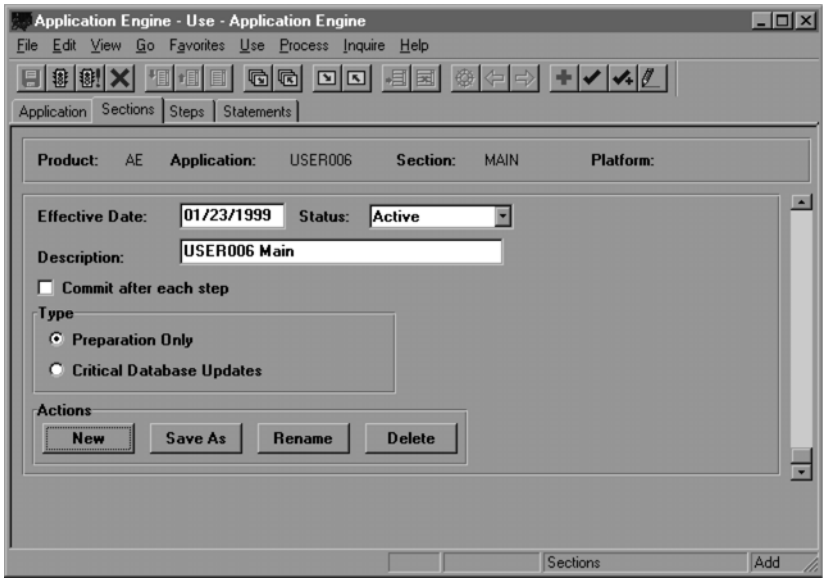


Figure 41.3 Defining section MAIN

We call our step STEP1 and click on the DO section radio button. Next, you click on the Edit button to indicate the section to perform.

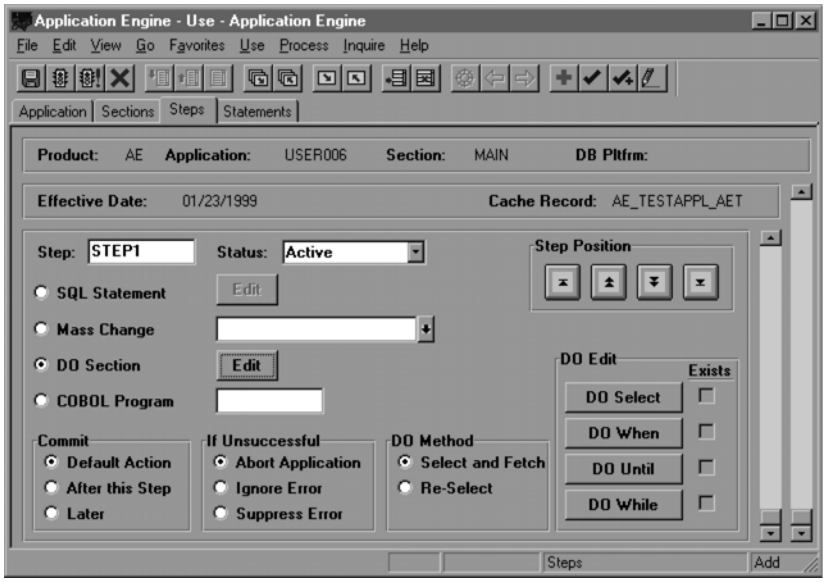


Figure 41.4 Defining STEP1

41.1.2 The &SECTION symbolic

When the DO Section Properties box appears, click on the dynamic DO checkbox (figure 41.5). Notice the `&SECTION` symbolic appears. You can also notice the product, application, and section edit boxes have been grayed out. This means the section you want to perform must exist within your Application Engine program. You cannot dynamically call a section from another Application Engine program.

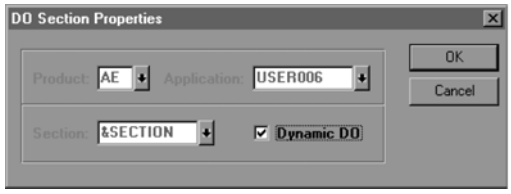


Figure 41.5
Designating dynamic section on DO Section Properties

When you return to the Step Definition panel, you'll notice the section being called is set to (DYNAMIC). When this step is executed, the DO section is determined by substituting the contents of the AE_SECTION cache field.

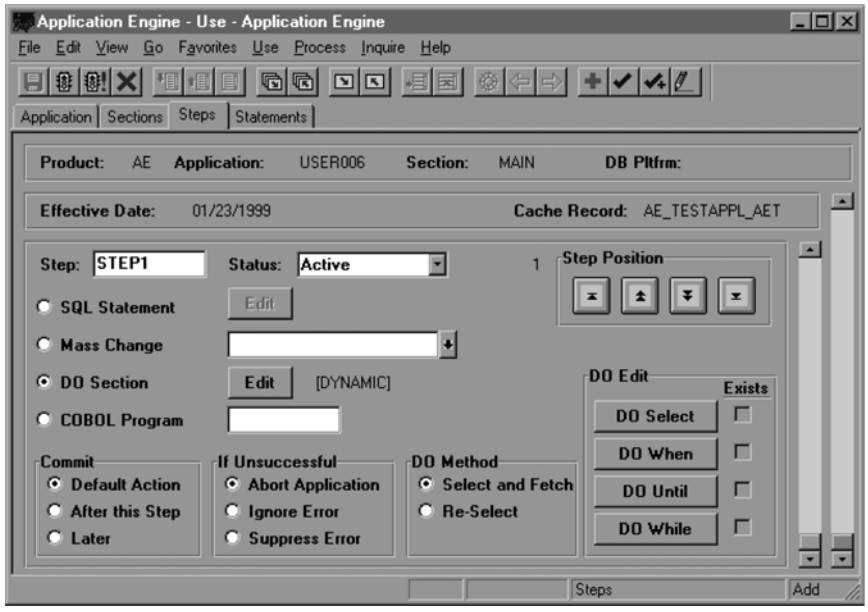


Figure 41.6 The DO section has been completed

We'll now add another section, called HELLO, to our application (figure 41.7).

Navigation: Go →PeopleTools →Application Engine →Use →Application Engine → Section →Add

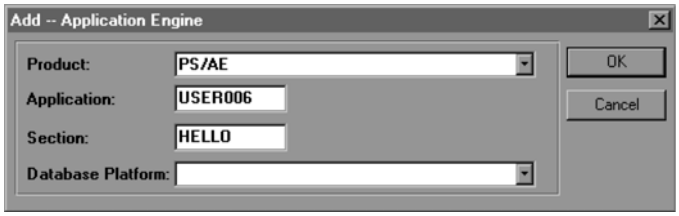


Figure 41.7 Adding another section called HELLO

Fill in the description for the HELLO section (figure 41.8).

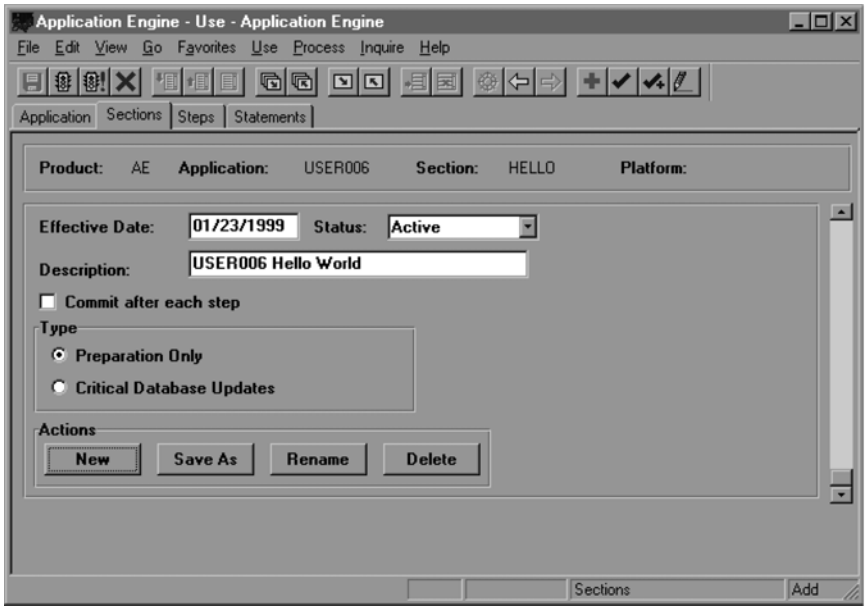


Figure 41.8 Defining section HELLO

We call the first step of the HELLO section STEP1 (figure 41.9).

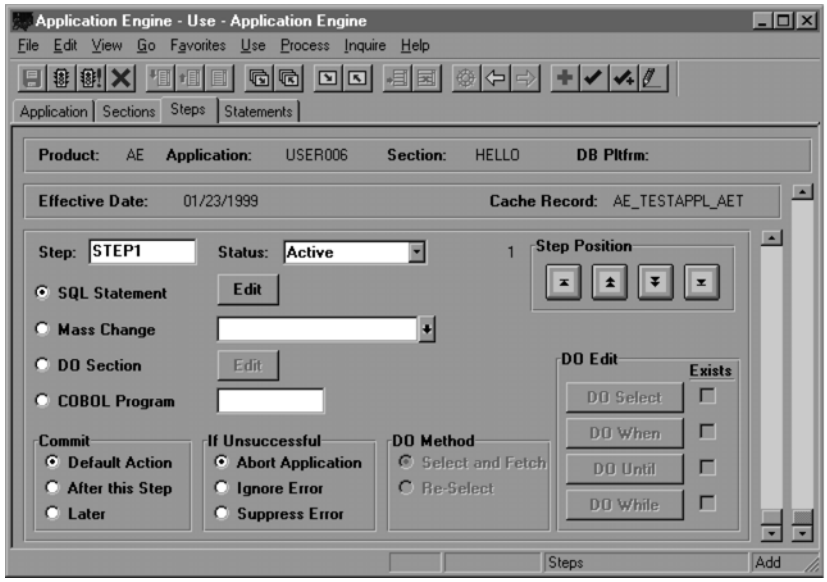


Figure 41.9 Defining STEP1 of section HELLO

We use the `&MSG` function to display "Hello World" on the message log (figure 41.10).

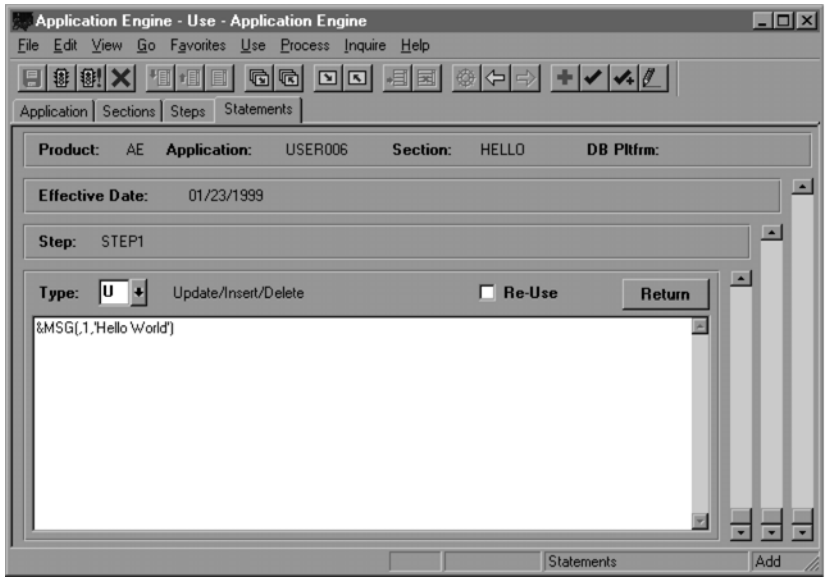


Figure 41.10 Adding the &MSG statement text

We'll now add another section to our application called GOODBYE (figure 41.11).

Navigation: Go →PeopleTools →Application Engine →Use →Application Engine → Section →Add

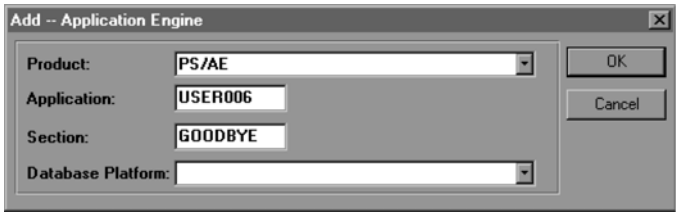


Figure 41.11 Adding another section called GOODBYE

Fill in the description for the GOODBYE section (figure 41.12).

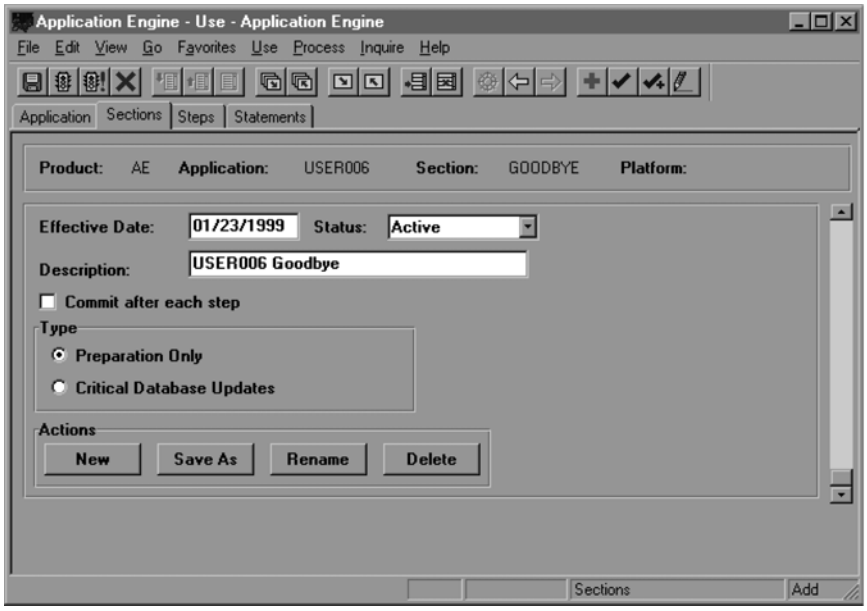


Figure 41.12 Defining section GOODBYE

We call the first step of the GOODBYE section STEP1 (figure 41.13).

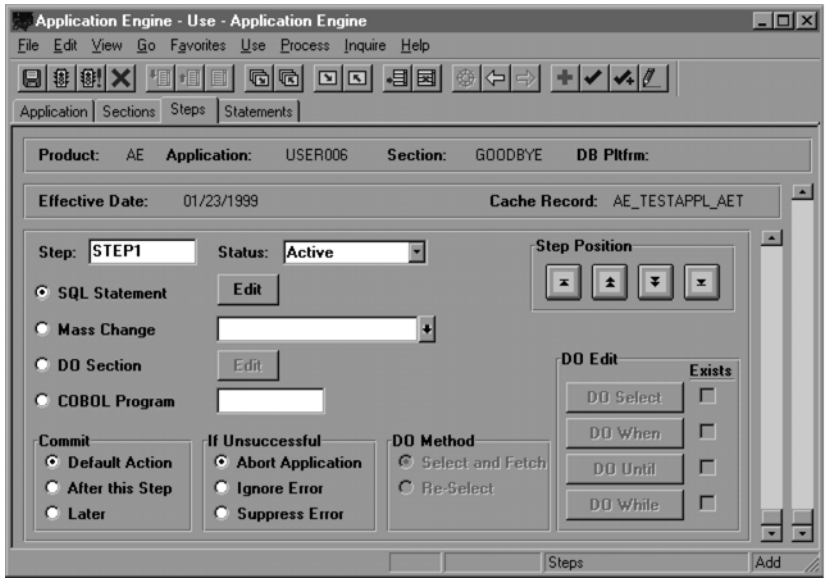


Figure 41.13 Defining STEP1 of section GOODBYE

We use the &MSG function to display "Goodbye" on the message log (figure 41.14).

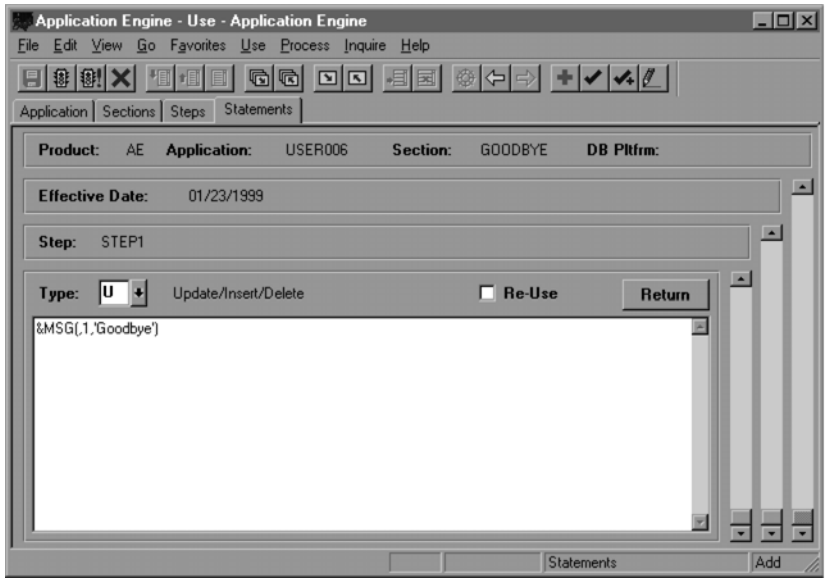


Figure 41.14 Adding the &MSG statement text

We're ready to test the USER006 application.

Return to the Process Request panel and add the Run Control ID `#USER006`. Click the OK button.

Navigation: Go →PeopleTools →Application Engine →Process →Request →Request →Add

Figure 41.15
Adding the Run Control ID

41.1.3 The AE_SECTION cache field

When the Process Request panel appears, click on the Field edit box. and scroll through the field list (figure 41.16). Select the cache field AE_SECTION.

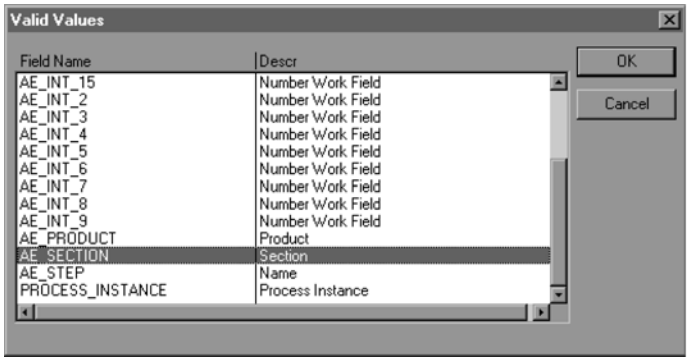


Figure 41.16 The cache field drop-down list box

We can now enter the name of the section which we'd like to perform. For the AE_SECTION cache field, we assign a value of HELLO. Our USER006 Application Engine program substitutes the section HELLO when it processes the `&SECTION` symbolic. Once the Process Request panel is populated correctly, click on the Traffic Signal to initiate a Process Scheduler request.

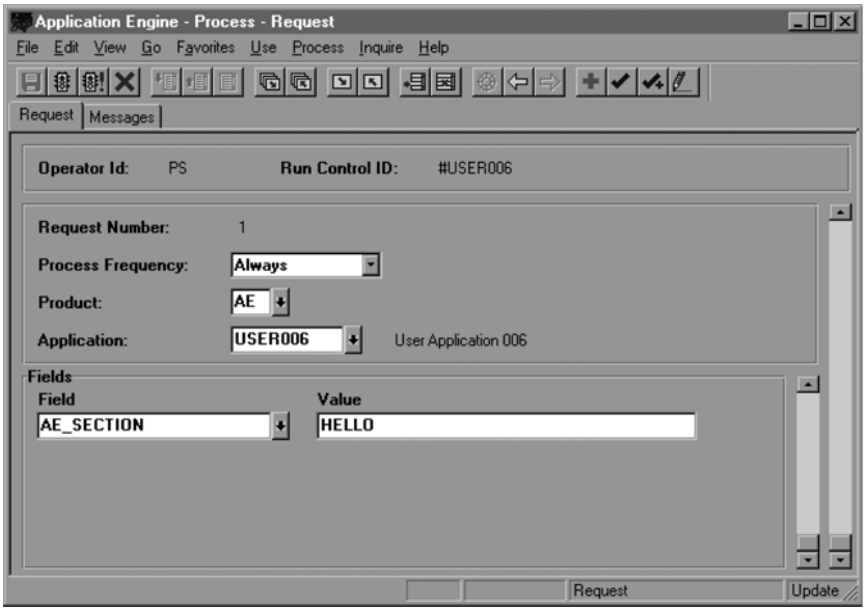


Figure 41.17 Assigning the dynamic section field the section HELLO

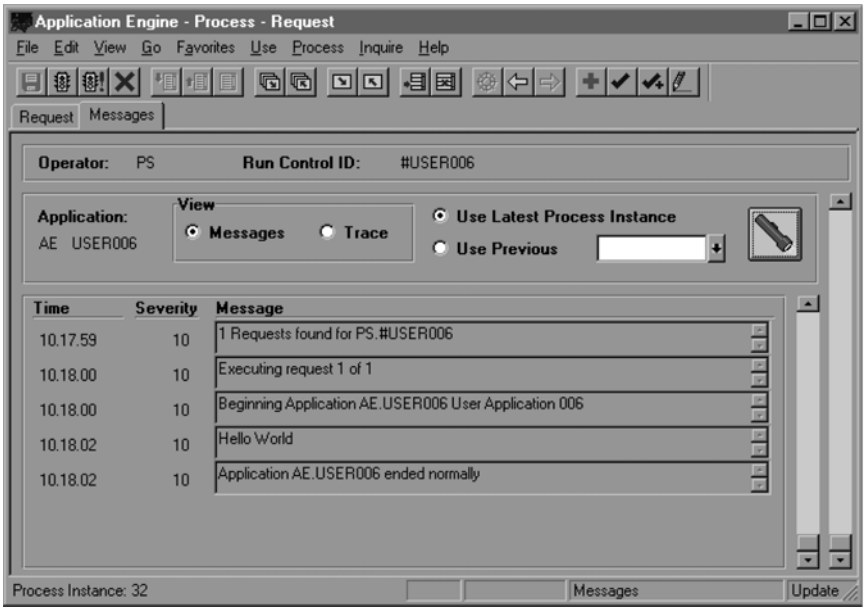


Figure 41.18 Reviewing process request messages

After examining the message log, you can see the HELLO section was performed. This was caused by populating the AE_SECTION cache field with the section which you'd like to perform.

Let's test our program again using another section. Return to the Process Request panel and assign the value GOODBYE to the AE_SECTION cache field (figure 41.19). Execute the program again and go to the message log to view the results.

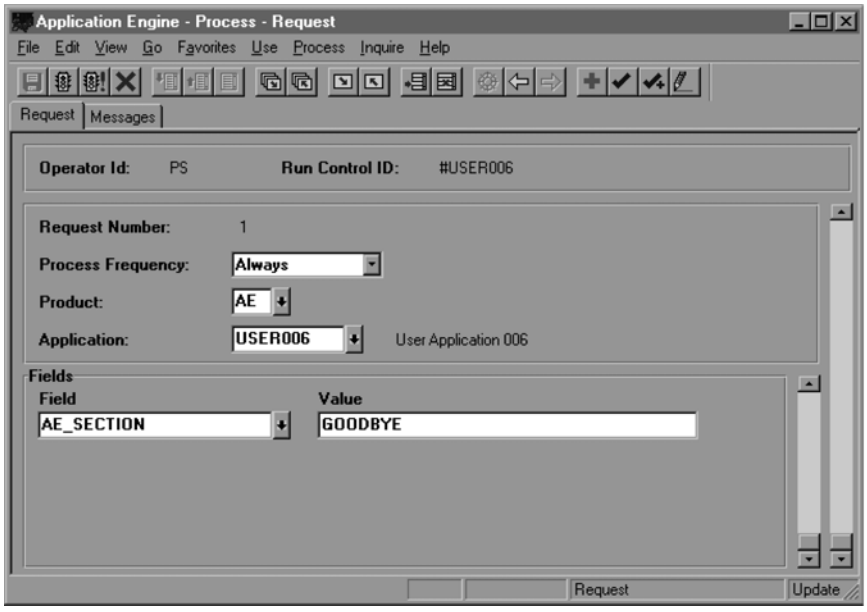


Figure 41.19 Assigning the dynamic section field the section GOODBYE

This time the GOODBYE section was performed (figure 41.20).

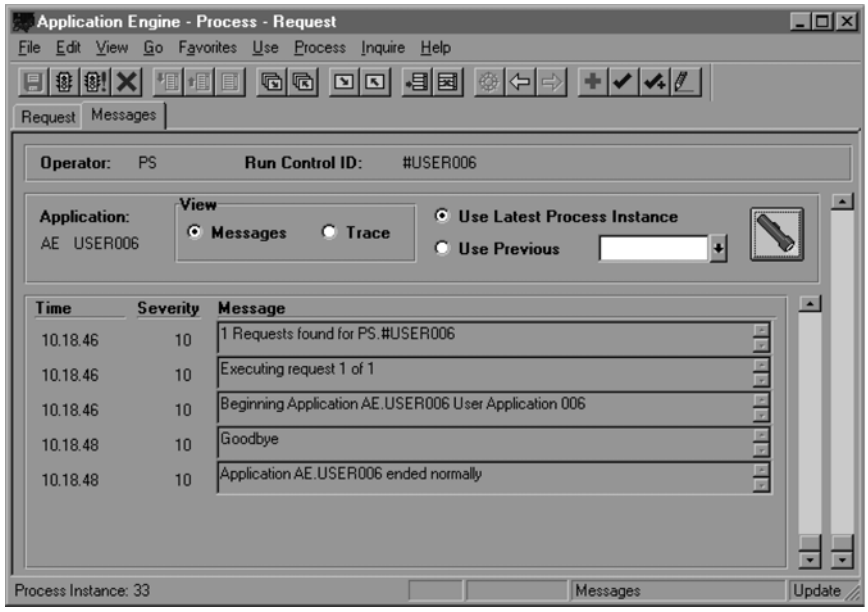


Figure 41.20 Reviewing process request messages

41.1.4 Multiple process requests

Let's try another quick experiment. We can use the Process Request panel to run multiple requests at once. We could have easily run the HELLO and GOODBYE versions of our exercise one after the other in the same run request.

Let's start by creating a new process request under a new Run Control ID.

Since we're running multiple requests, we use the Run Control ID 'MULTIPLE' (figure 41.21). Now we need to populate the Process Request panel.

Navigation: Go →PeopleTools →Application Engine →Process →Request →Request →Add

Figure 41.21 Adding the Run Control ID

So far nothing seems different (figure 41.22). The dynamic section HELLO will be executed. Take a look at the outermost scroll bar on the right. Each process request we enter can be viewed using the outer scroll bar. We can add another process request by placing the cursor in one of the outer scroll fields (process frequency, product, or application) then pressing the F7 key to insert a new row.

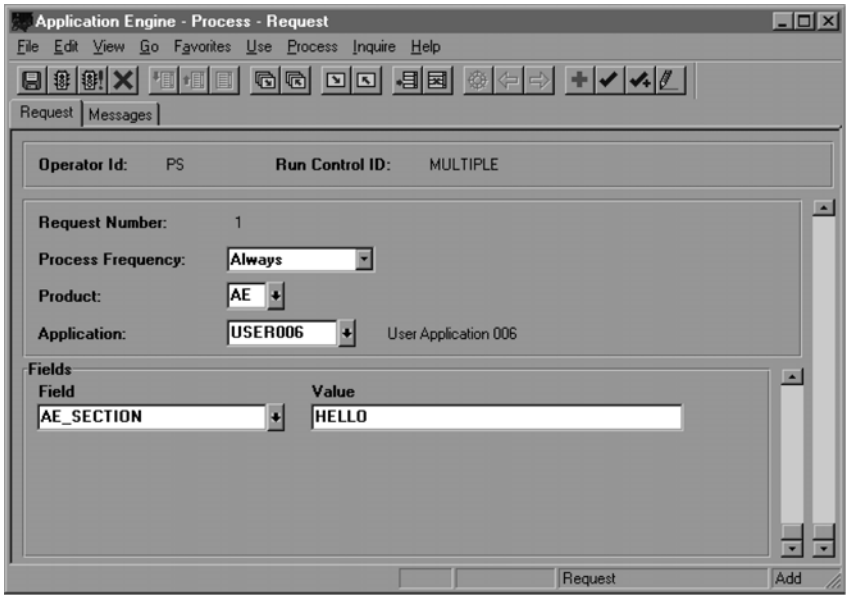


Figure 41.22 Adding Request Number 1 using the HELLO dynamic section

After pressing the F7 key, a new row can be filled in with our second set of run parameters. In figure 41.23, we have selected the dynamic section GOODBYE. Notice the request number for the GOODBYE section is incremented to 2. When we submit our request, the HELLO request will be executed followed by the GOODBYE request. Let's run the request and look at the message log.

Figure 41.24 displays the Messages panel. The first line shows that two requests were found for our run. The first request is executed (1 of 2) and displays the "Hello World" message. The second request is then executed (2 of 2) and displays the "Goodbye" message (though not visible in the screen shot). We could have executed all of our exercises consecutively in one process request run.

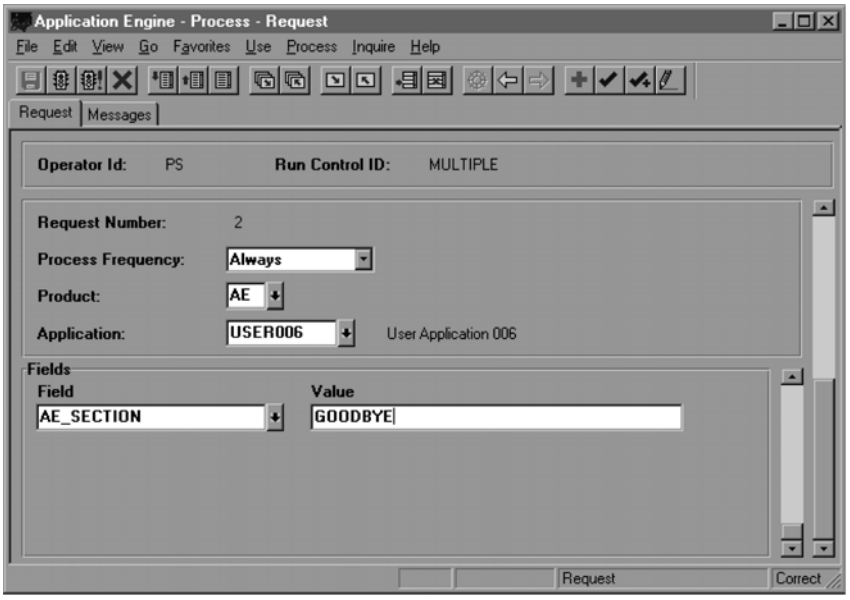


Figure 41.23 Adding Request Number 2 using the GOODBYE dynamic section

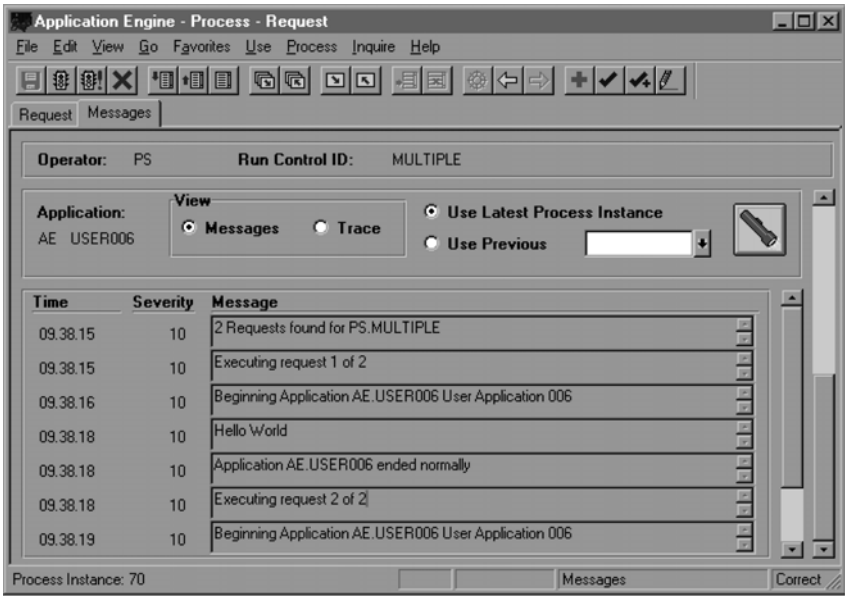


Figure 41.24 Examining the message log for the multiple request run

41.2 SQR/APPLICATION ENGINE COMPARISON

If we look at the logical structure of both our programs now

SQR:

```
Begin-Program
        User prompted
        Main-Step1
        When 1
            Hello-Step1
        When 2
            Goodbye-Step1
```

Application Engine:

```
USER001
    Cache assignment
    MAIN.STEP1
            MAIN.&SECTION
```

We can see the Application Engine program is much more streamlined. The section is assigned on the Process Request panel and used in place of the `&SECTION` symbolic.

41.3 DYNAMIC SECTIONS IN PEOPLESOFT

You may be wondering where you can find an example of dynamic sections in an existing PeopleSoft application. A perfect example would be the payment predictor process, called PREDICT, found in Accounts Receivable. Its purpose is to match incoming payments with the associated items (or invoices). Several delivered sections or algorithms exist that can be selected to match payments based on certain criteria. A special payment predictor setup table is used to store the algorithm name to be used. When PREDICT is run, the setup information is retrieved. The algorithm is assigned to the AE_SECTION cache field and substituted for the `&SECTION` symbolic. Using dynamic sections in this manner provides a great deal of flexibility.

KEY POINTS

1. Sections are called dynamically when the value found in the `AE_SECTION` bind variable is substituted as the section represented by the `&SECTION` symbolic.
2. Dynamic sections can be found in several PeopleSoft Applications such as Payment Predictor. The use of dynamic sections allows the user to tailor programs to meet their own business requirements.
3. Multiple process requests can be submitted in one execution run. This is useful when you would like processes to run consecutively.

CHAPTER 42

Using Run Controls—part A

The exercises we have completed thus far were designed to demonstrate the capabilities of Application Engine. You may not have a need to display a message saying "Hello World." You may not need any of the programs we've created! The important thing is that you've learned the concepts behind Application Engine development. We're now ready to produce something of value. In chapter 28, we mentioned that PeopleSoft does not provide a tool to delete obsolete process definitions. You can only delete these outside of PeopleSoft using your native SQL tools. Let's create a utility to accomplish this using Application Engine. This is the perfect opportunity to introduce Run Control records in Application Engine. In order to implement this utility, we're going to go through the complete cycle of program development. Let's get started.

42.1 EXERCISE 7: DELETE PROCESS DEFINITIONS

Let's refresh our memory first. In chapter 28, we listed the SQL statements necessary to clean up the process definition tables.

Let's look at the SQL statements used to remove the MYPROB01 SQR Report process definition from all associated tables:

```
DELETE
  FROM PS_PRCSDEFN
 WHERE PRCSNAME = 'MYPROB01'
   AND PRCSTYPE = 'SQR Report';

DELETE
  FROM PS_PRCSDEFNGRP
 WHERE PRCSNAME = 'MYPROB01'
   AND PRCSTYPE = 'SQR Report';

DELETE
  FROM PS_PRCSDEFNPNL
 WHERE PRCSNAME = 'MYPROB01'
   AND PRCSTYPE = 'SQR Report';

DELETE
  FROM PS_PRCSDEFNXFER
 WHERE PRCSNAME = 'MYPROB01'
   AND PRCSTYPE = 'SQR Report';

DELETE
  FROM PSPRCSRQST
 WHERE PRCSNAME = 'MYPROB01'
   AND PRCSTYPE = 'SQR Report';

DELETE
  FROM PSPNLFIELD
 WHERE PRCSNAME = 'MYPROB01'
   AND PRCSTYPE = 'SQR Report';
```

In our new process, we'll allow the user to enter the PRCSNAME and PRCSTYPE on a new Run Control panel. The Application Engine process we develop will remove the process definition from the six tables listed. Notice the first four tables use the standard prefix `'PS_'` while the last two tables do not. We'll make our program interesting by accounting for this in our program.

Before we proceed, let's take a look at the development requirements. When developing applications, this is a critical and often overlooked step. Let's go over the steps we need to take to produce our application:

42.1.1 Application development steps

- create a custom Run Control record
- add PeopleCode to the Run Control record
- create a custom panel
- create a custom panel group
- attach the panel group to a menu
- assign operator security to the menu item
- create a process definition for our Application Engine program
- create the Application Engine program
- test our new application

We have a lot of work ahead of us. Fortunately, this will be a fairly easy task using PeopleTools.

One thing I'd like to resolve now is the name of our Application Engine program. This means the combination of product and application ID. We will use the product A/E (as we've done in all our exercises) and the name of the application is going to be MYPRCSDL.

Let's start by building a new Run Control record.

42.2 BUILD A NEW RUN CONTROL RECORD

Application Engine programs use a primary Run Control record called AE_REQUEST. You may not have realized it at the time, but this is the underlying Run Control record we've been using when testing our applications in exercises 1 through 6. We're going to create a new custom Run Control record that will be linked (as a child record) to the AE_REQUEST record. We start by cloning the AE_REQUEST record (figure 42.1).

Our Run Control record only needs five fields. Of course, the record needs the standard AE_REQUEST keys, which are OPRID, RUN_CNTL_ID, and REQUEST_NBR. In addition, we want the user to enter the process type and process name. We'll now clone the AE_REQUEST record for our purposes. Open the record AE_REQUEST and remove all fields except OPRID, RUN_CNTL_ID and REQUEST_NBR. Next, add the fields PRCSTYPE and PRCSNAME. The result should look like figure 42.1. Be careful not to save the record using the AE_REQUEST name! We save it under a new name: MY_RUN_CNTL_AE.

Navigation: Go →PeopleTools →Application Designer →File →Open →
Record →AE_REQUEST

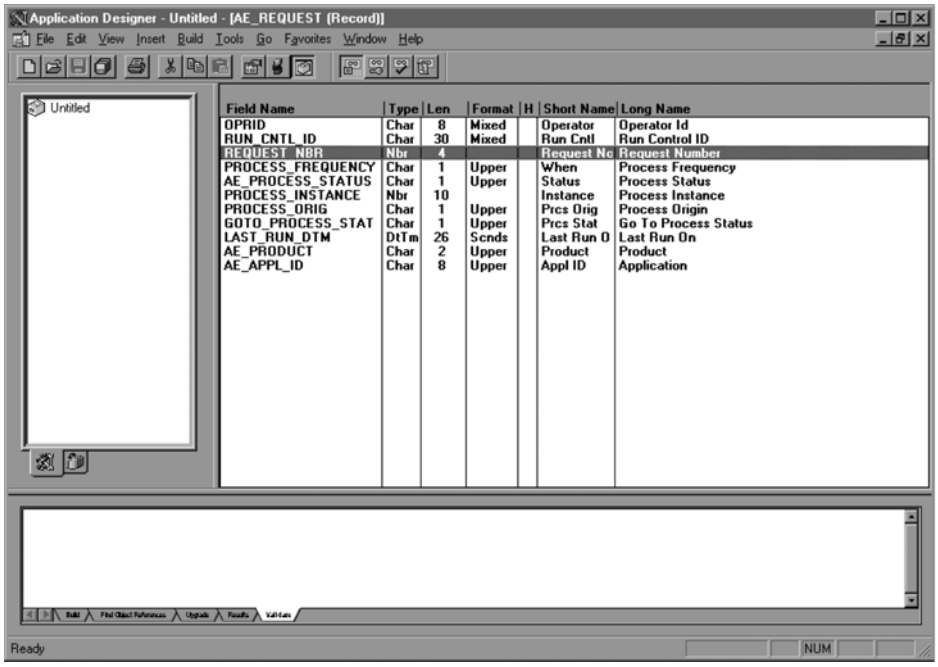


Figure 42.1 Cloning the AE_REQUEST Run Control record

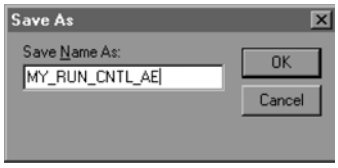


Figure 42.2 Saving our Run Control record

Use File →Save As to save the record under a new name. Figure 42.2 shows the prompt box filled in with our new name. Let's add some underlying edit prompts for our two new fields.

Figure 42.3 displays the record in Edit View. We've added the PRCSTYPE_VW and PRCSDEFN records as edit tables for the PRCSTYPE and PRCSNAME fields. This allows the user to select the process type and process name on the Run Control panel using drop-down lists.

Also, note the record keys for our new Run Control record are OPRID, RUN_CNTL_ID, and REQUEST_NBR. The key attributes were copied when we saved the AE_REQUEST record under our new record name.

Because we plan to integrate our Run Control record with the AE_REQUEST record, we have to resolve a couple of issues. This will become clearer when we create the panel. For now, let's look at the required fields on the AE_REQUEST record.

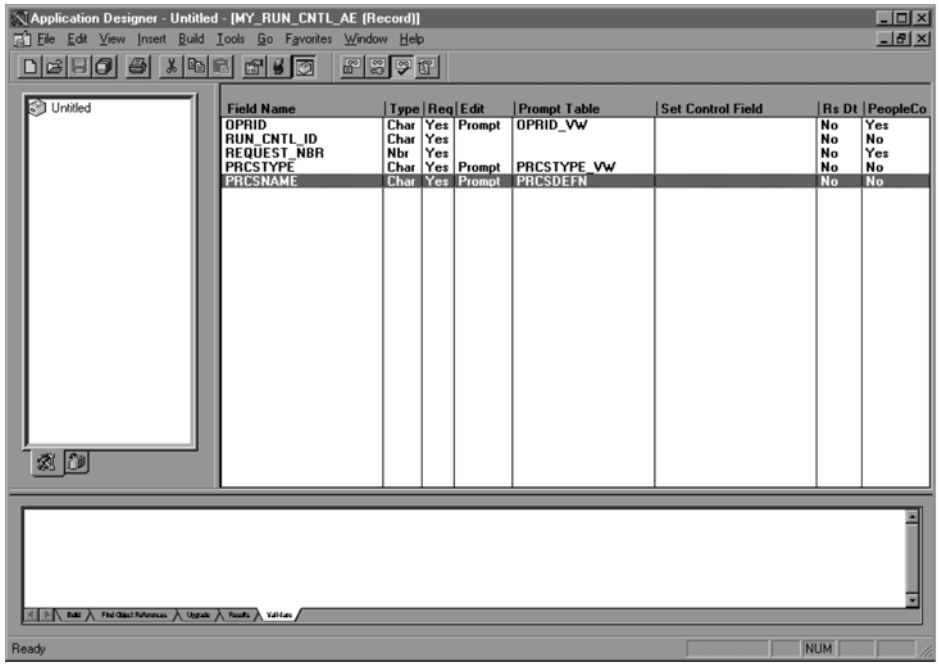


Figure 42.3 Adding edit prompts to the process type and process name fields

Figure 42.4 displays the required fields on the AE_REQUEST record. The first three are the record keys and will be filled in with the OPRID, RUN_CNTL_ID, and REQUEST_NBR. The last two fields, AE_PRODUCT and AE_APPL_ID, are also required. We plan to use the AE_REQUEST record as the primary record on our panel with our new custom record placed in the panel as a child record. We will not be able to save the record without a product or application ID. This is an easy customization using PeopleCode. We'll simply initialize these two fields with the product and application ID we're going to use. We've already decided to use AE.MYPRCSDL as the product and application ID. Let's add some PeopleCode.

Figure 42.5 shows the PeopleCode to populate the required fields AE_PRODUCT and AE_APPL_ID in the AE_REQUEST record.

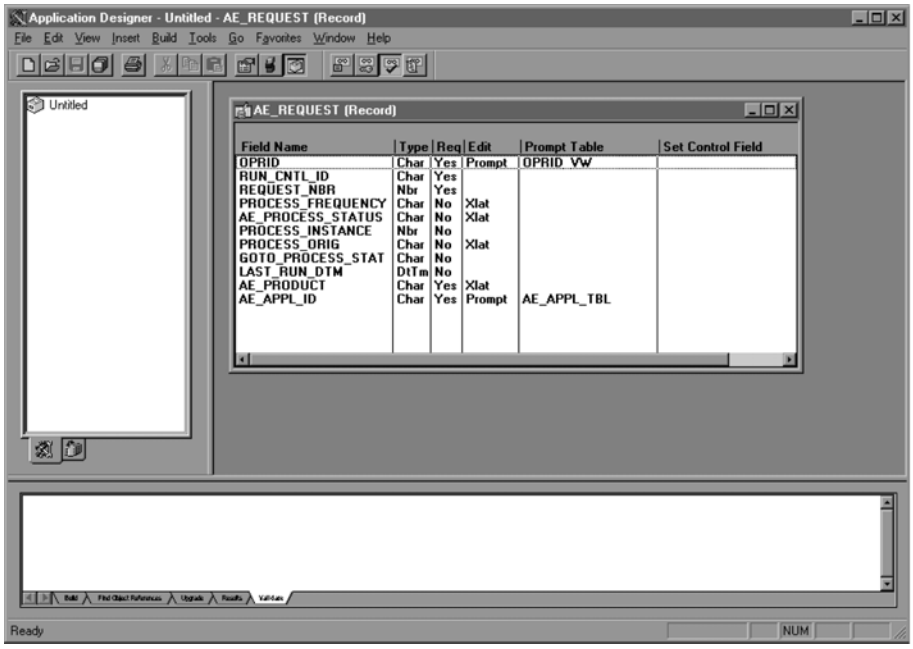


Figure 42.4 Looking at the AE_REQUEST required fields

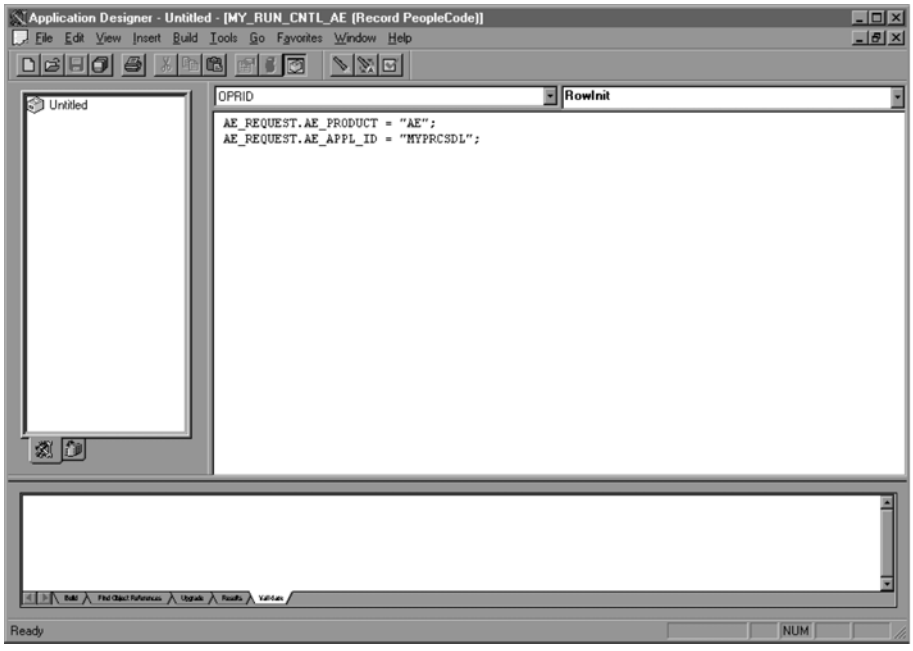


Figure 42.5 Updating AE_REQUEST record with our product and application ID

Build the current object using SQL `create`. This will create the table at the database level. We'll now make some modifications to our cache record, USER_AET.

42.2.1 Modify our existing cache record

The cache record USER_AET is the same record we've used for most of our exercises. We can re-use this by adding the additional fields we need for our application. We need to add the process type and process name fields passed from the Run Control record. We'll also add the AE_SECTION field to allow us to execute sections dynamically. We'll explain this as we develop the application.

Figure 42.6 shows the modifications we've made to the USER_AET cache record. As you can see, the three fields have been added. Now, we need to build the current record object using SQL `create`. We can now build the Run Control panel.

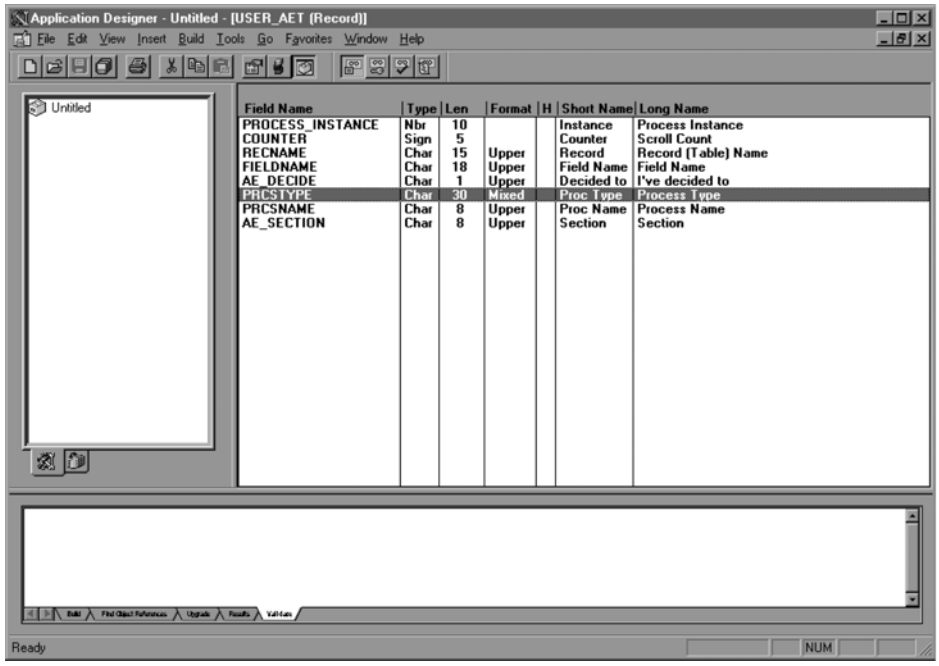


Figure 42.6 Modifying the USER_AET cache record

42.3 BUILDING THE RUN CONTROL PANEL

We can clone an existing panel used specifically for Application Engine. The panel name is AE_REQUEST. We used this panel when testing our applications from previous exercises. We're going to remove most of the fields on the panel, then add the new ones from our custom Run Control record.

Figure 42.7 shows the AE_REQUEST panel with all the fields intact. Let's remove all fields except operator ID, Run Control ID, and request number. We also will keep the rightmost scroll bar. Then we'll add the Process Type and Process Name fields from our custom Run Control record. We'll make a few slight adjustments that may seem a bit odd at first, but we'll explain as we go.

Navigation: Go →PeopleTools →Application Designer →File →Open →Panel →AE_REQUEST

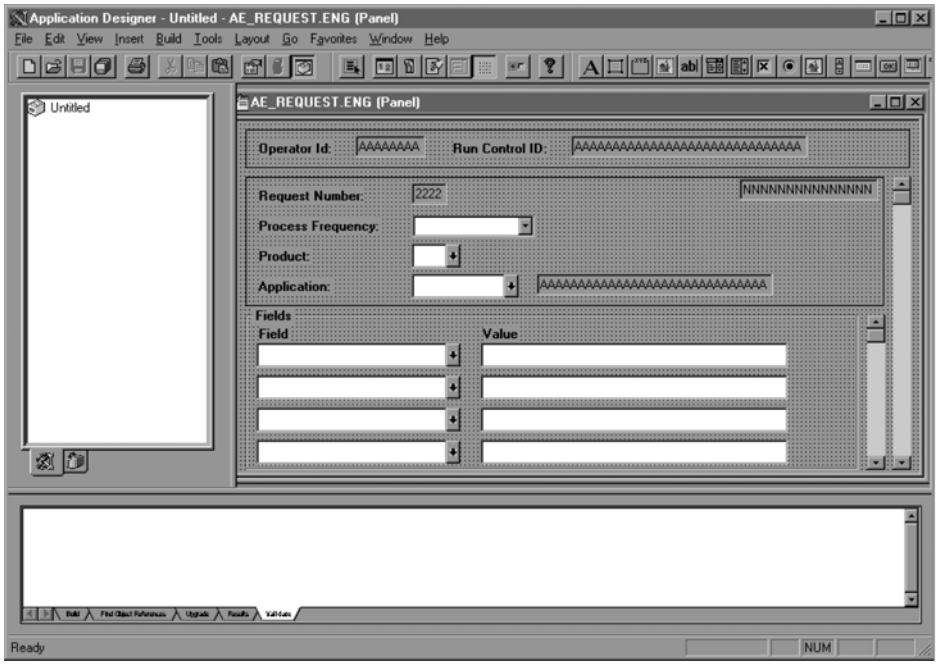


Figure 42.7 Cloning the AE_REQUEST panel

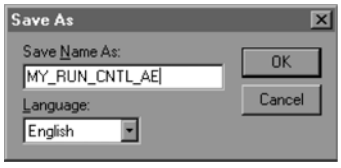


Figure 42.8 Saving the panel with our new name

Let's save the panel using the name MY_RUN_CNTL_AE (figure 42.8).

Our panel is now complete.

NOTE When cloning, make sure you don't inadvertently save the object under its original name. This is true for all cloned objects—records, panels, and so on.

Figure 42.9 shows the completed panel named MY_RUN_CNTL_AE. We've made some cosmetic adjustments as well. We moved the process request number into the top frame. We also surrounded the user fields Process Type and Process Name with a group box and labeled it "Processing Parameters." Also, notice the inner scroll bar. This contains the fields for our custom record. We had no choice in this matter. PeopleSoft does not allow you to place multiple records under the same scroll bar. We are going to change the inner scroll bar properties so it is not visible to the user.

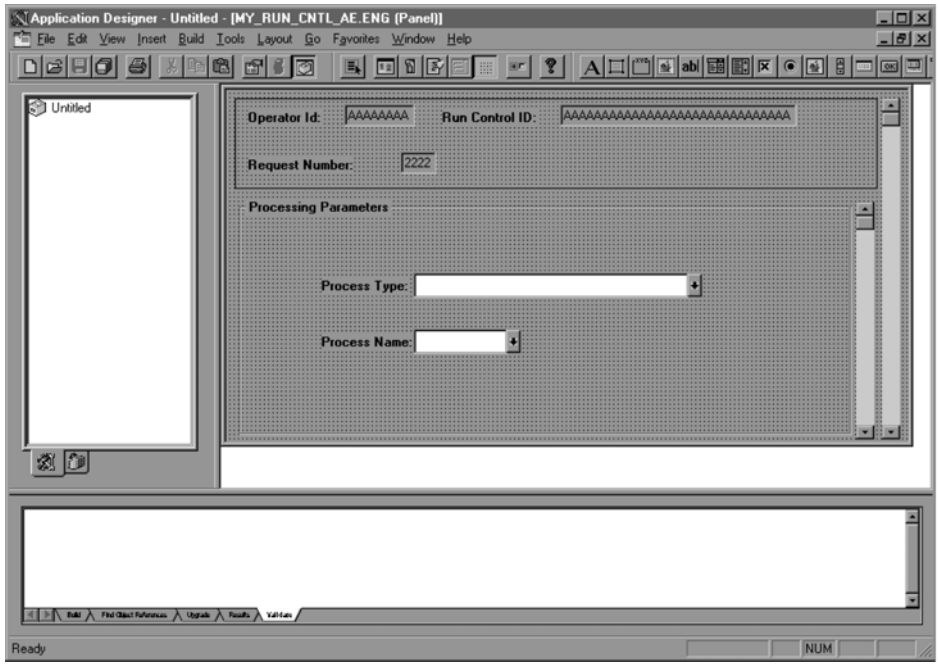


Figure 42.9 Our new Run Control panel is complete

In our previous exercise, we demonstrated the ability to submit multiple process requests. This is one of the features of the AE_REQUEST panel (which we cloned). For our purposes, we do not want to allow multiple process requests. Deleting process definitions can be considered a dangerous function. Restricting this function to delete one process definition at a time is a wise decision.

Because the only remaining fields within the outer scroll are "Display Only," we cannot insert an additional process request. This is the desired effect. We could have set the outer scroll properties to restrict rows from being inserted and to also make the scroll bar invisible. We'll leave the outer scroll bar properties as they are. Because the inner scroll bar has data entry fields, we have to modify the inner scroll bar properties.

Figure 42.10 displays the inner scroll bar properties we've set. The Occurs count is 1 so only one row may exist as a child of the AE_REQUEST record. The scroll bar will be invisible. The user will not be able to insert or delete rows within the inner scroll bar (via F7 and F8).

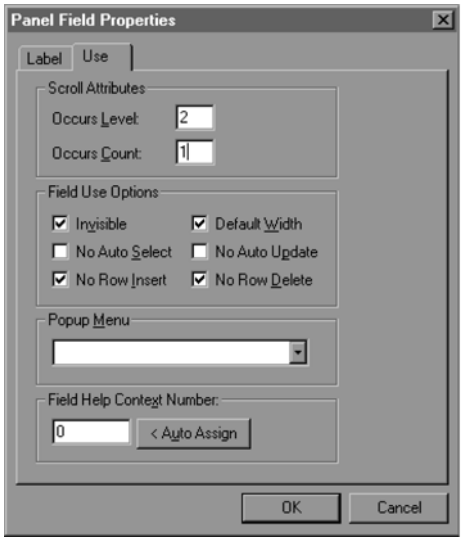


Figure 42.10
Inner scroll bar properties

Figure 42.11 shows the panel elements that make up the entire panel. You can access this screen by selecting Layout →Order on the Application Designer menu bar. Make sure your panel entries match those in figure 42.11.

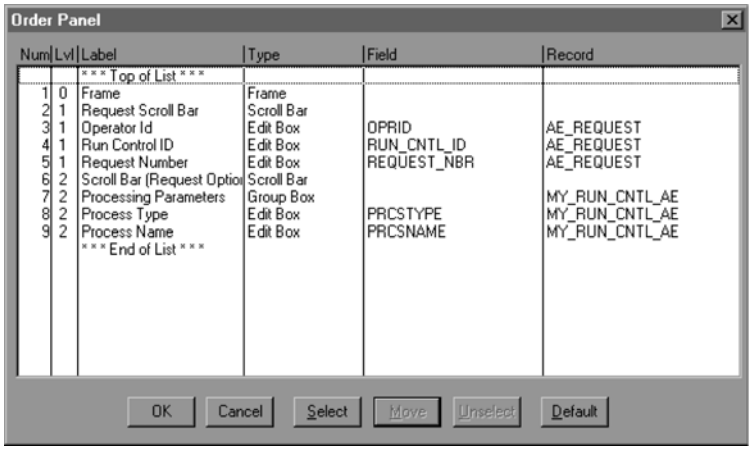


Figure 42.11 The MY_RUN_CNTL_AE panel elements

42.4 CREATE A NEW PANEL GROUP

Now, let's create a panel group for our new Application Engine process.

Figure 42.12 shows the panels we've added to our new panel group. Of course, the custom Run Control panel has been added. We've also added the panel AE_MESSAGE_LOG. You've seen this panel during our exercises to look at Application Engine messages. We'll attach it to our panel group so we have a convenient means of viewing messages. We've also entered a descriptive label for each panel in the Item Label column.

Navigation: Go →PeopleTools →Application Designer →File →New →PanelGroup

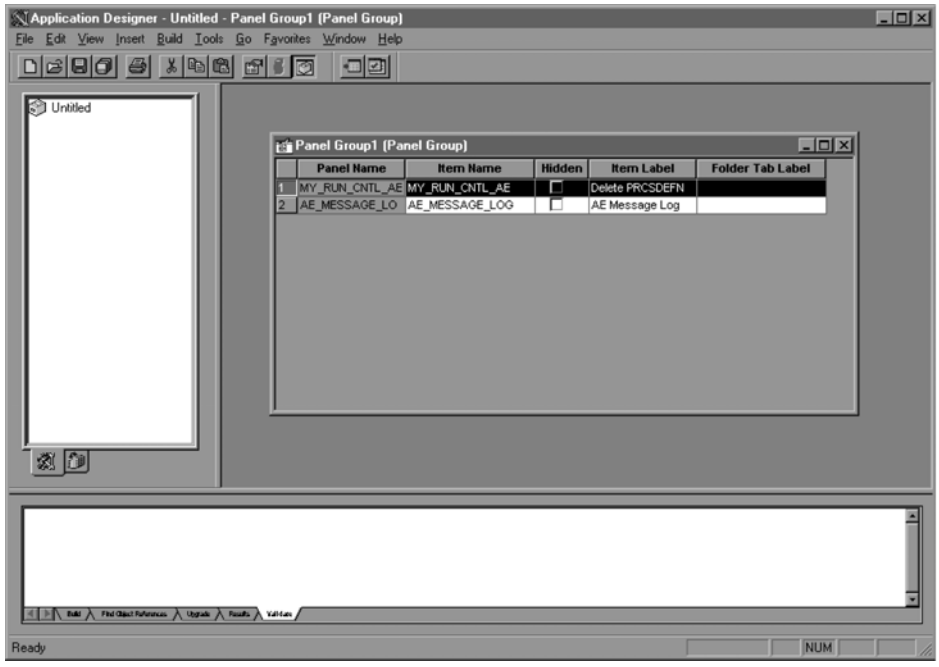


Figure 42.12 Creating the new panel group

NOTE Much of the synchronization between the AE_REQUEST record and our MY_RUN_CNTL_AE record was made in anticipation of adding the AE_MESSAGE_LOG panel to our new panel group. It's always a nice touch to give the user access to Application Engine messages from the same panel group.

Before we can save the panel group, we need to enter the panel group properties. You can access the properties by clicking the right mouse button or pressing ALT-ENTER. Figure 42.13 shows the description added for our panel group.

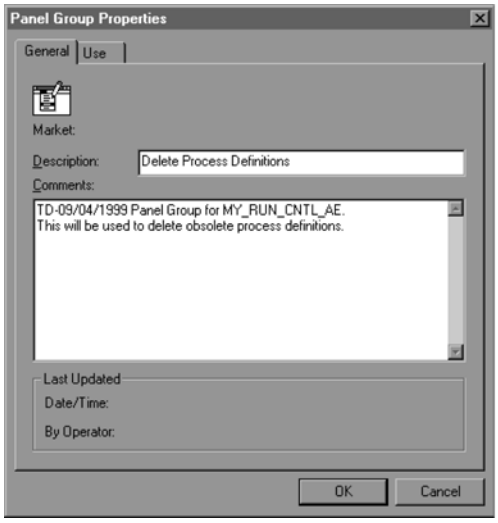


Figure 42.13
Adding a description to the Panel Group Properties

We've added AE_REQUEST as the search record in figure 42.14. We've also checked the `Add` and `Update/Display` actions. We're ready to save our panel group.

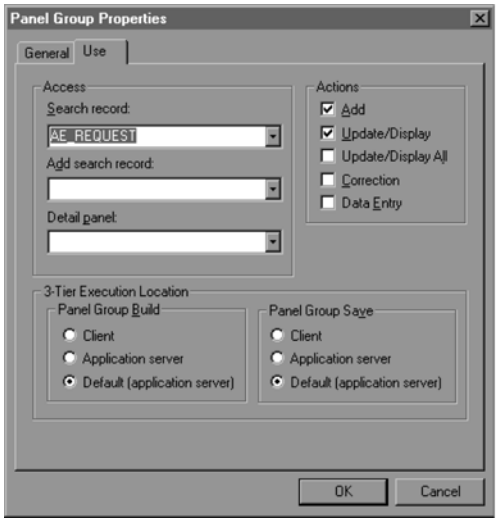


Figure 42.14
Adding a search record to the Panel Group Properties

Figure 42.15 Saving our new panel group

Figure 42.15 shows the panel group as it's being saved. We'll use the name MY_RUN_AE.

Our next step is to add the panel group to an existing menu. Let's take a step back and review what our process actually does. When the user enters a process type and process name, the application will physically remove all references to it without a trace. If the wrong process type/name is entered it will be deleted! A menu with limited authorization would be suitable. Only a select few should be running this process. Since this can be considered a PeopleTools utility, it seems logical to add this to the delivered PeopleTools utility menu.

42.5 ATTACHING THE PANEL GROUP TO A MENU

Let's attach the panel group to the PeopleTools utility menu.

Figure 42.16 shows the utilities menu. The column labeled "Process" is a perfect place for our new panel group. Let's update the menu item properties for the next available menu item position (the open rectangle).

Navigation: Go →PeopleTools →Application Designer →File →Open →Menu →UTILITIES

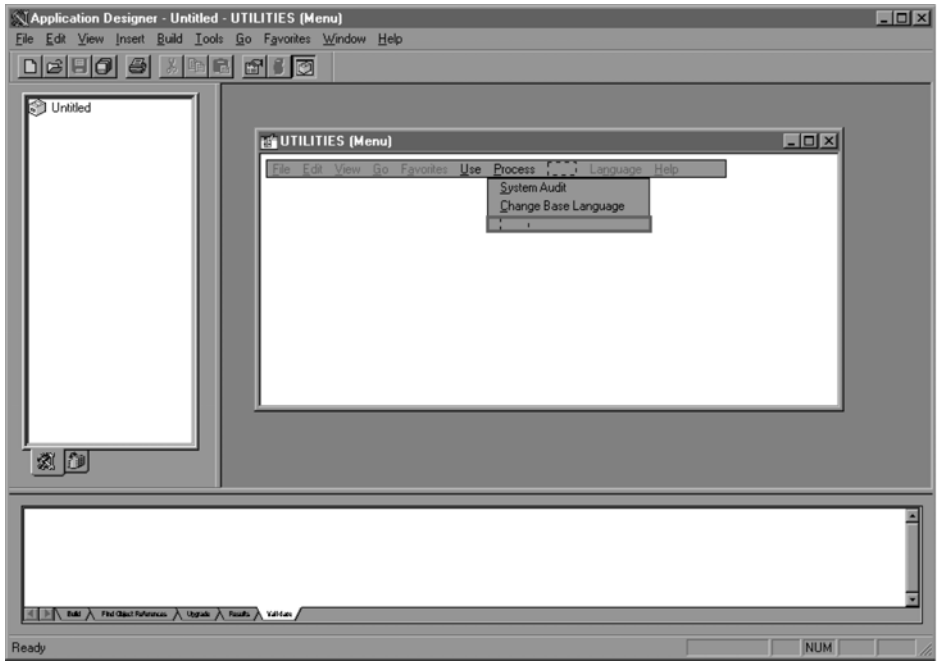


Figure 42.16 Adding the panel group to the Utilities menu

Enter the menu item properties as shown in figure 42.17 using the new panel group MY_RUN_AE. Use descriptive text for the menu item label. "Delete `Process Definition`" is a good choice.

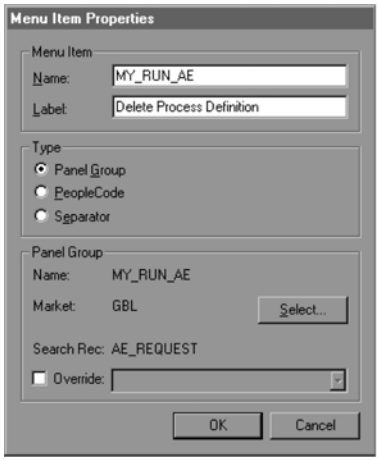


Figure 42.17
Entering the menu item properties

Now, click OK and save the utilities menu. Our next step is to assign security to the new menu item.

42.6 ASSIGNING OPERATOR SECURITY

We'll assign access to the operator class ALLPANLS. Remember, this process should be limited to a small group of people.

Figure 42.18 shows the security administrator panel. The UTILITIES menu is highlighted for the ALLPANLS operator class. Double-click the UTILITIES Menu to access our new menu item.

Navigation: Go →PeopleTools →Security Administrator →File →Open →ALLPANLS → Menu Items →UTILITIES

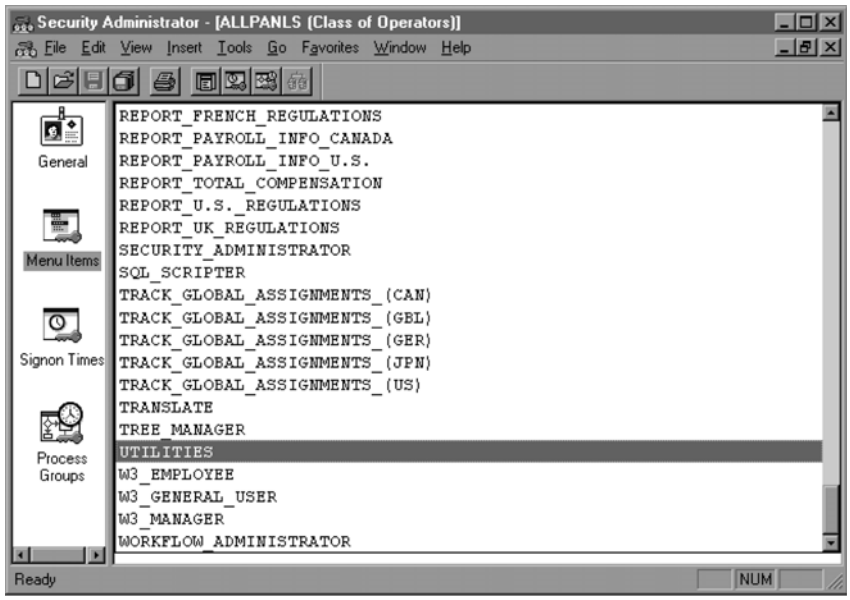


Figure 42.18 Assigning security to the ALLPANLS operator class

You can see our new menu item in figure 42.19. To assign security access to the ALLPANLS class, simply click on the associated MY_RUN_AE items. These are the last four items that appear. Once all four items are highlighted, click on the OK button.

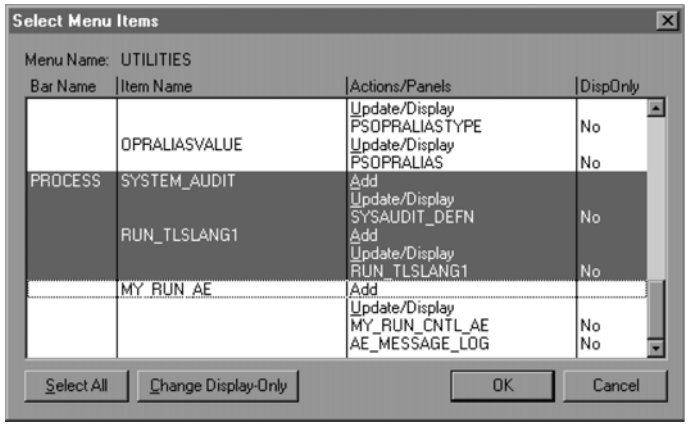


Figure 42.19 Our new menu item (MY_RUN_AE) as it appears in the menu

Now, save the new operator class settings (File →Save).

Let's sign off PeopleSoft and log back on so that our new security goes into effect.

42.7 TESTING THE NEW PANEL

Let's test the modifications we've made. We haven't actually created the Application Engine program yet, but we can see if our Run Control panel is behaving correctly.

We can immediately tell that our operator security changes were successful: the panel does appear in the menu. Using a test Run Control ID (MYTEST), we can successfully select any process type and process name from the drop-down lists (figure 42.20). An important test would be saving the Run Control record. In our case, the record was saved without a problem. Part of the reason for our successful result was due to the PeopleCode we put in place to populate the AE_PRODUCT and AE_APPL_ID fields in the AE_REQUEST record. This alleviated the required field constraint found on the AE_REQUEST record.

Navigation: Go →PeopleTools →Utilities →Process →Delete Process Definition → Add →MYTEST

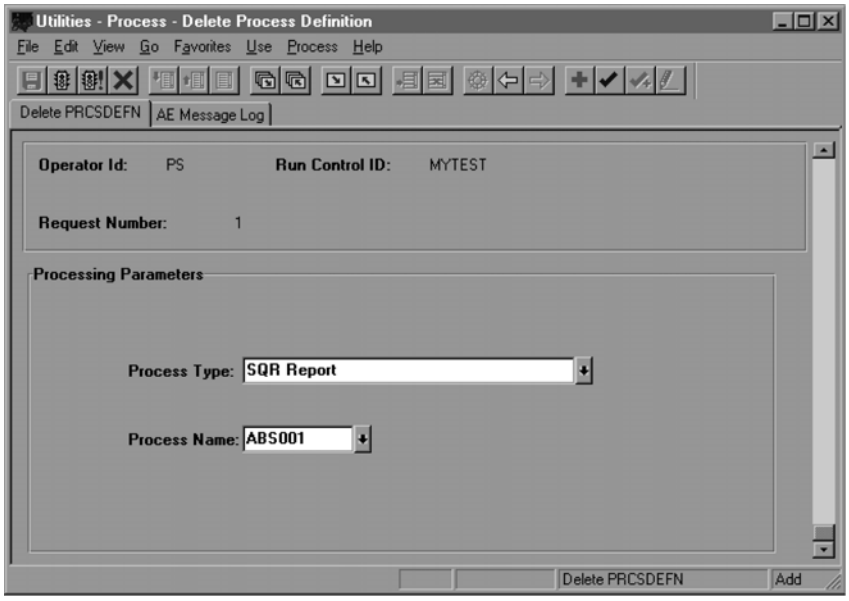


Figure 42.20 Our new menu item (MY_RUN_AE) as it appears in the menu

We can verify that the rows in both tables are being saved correctly using the database's query tool. Figure 42.21 shows the results of queries made against both tables.

Notice the second query has the product and application ID populated correctly. This was assigned by the PeopleCode we placed in the MY_RUN_CNTL_AE record. Everything seems to be working as planned.

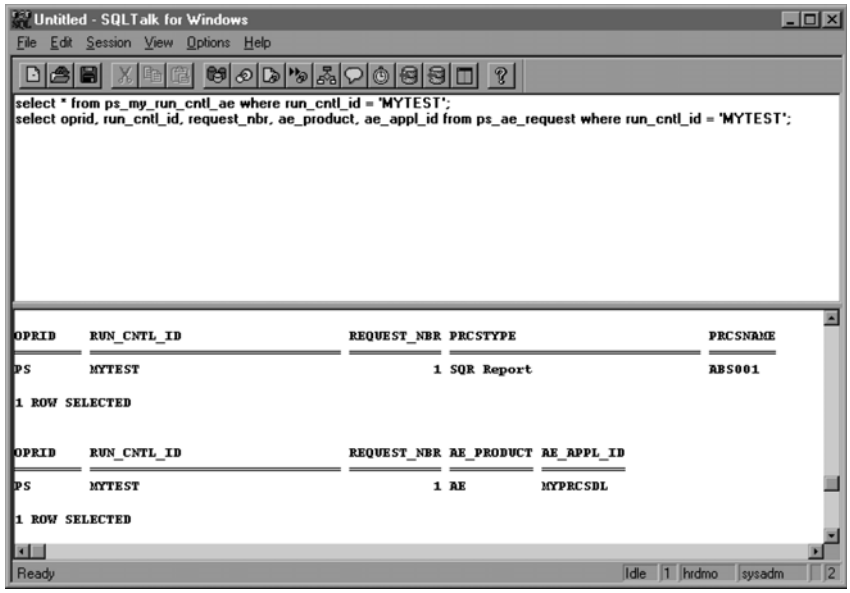


Figure 42.21 Looking at the resulting rows using SQL*Talk

Our next step is to create a process definition for our application.

42.8 CREATING OUR PROCESS DEFINITION

We'll now create the process definition for our application. The process type is Application Engine. The name of our application is MYPRCSDL.

We've added the process type and process name (figure 42.22). Let's enter the process definition information into Process Scheduler.

Navigation: Go →PeopleTools →Process Scheduler →Use →Process Definitions → Process Definitions →Add

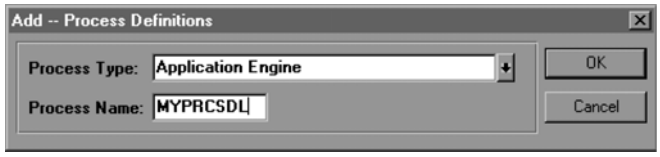


Figure 42.22 Adding our process definition

Figure 42.23 shows the process definition information we need to add. We've added our new MY_RUN_AE panel group to the definition screen along with additional items such as descriptive text and process security groups. The process class of Application Engine programs is COBOL SQL.

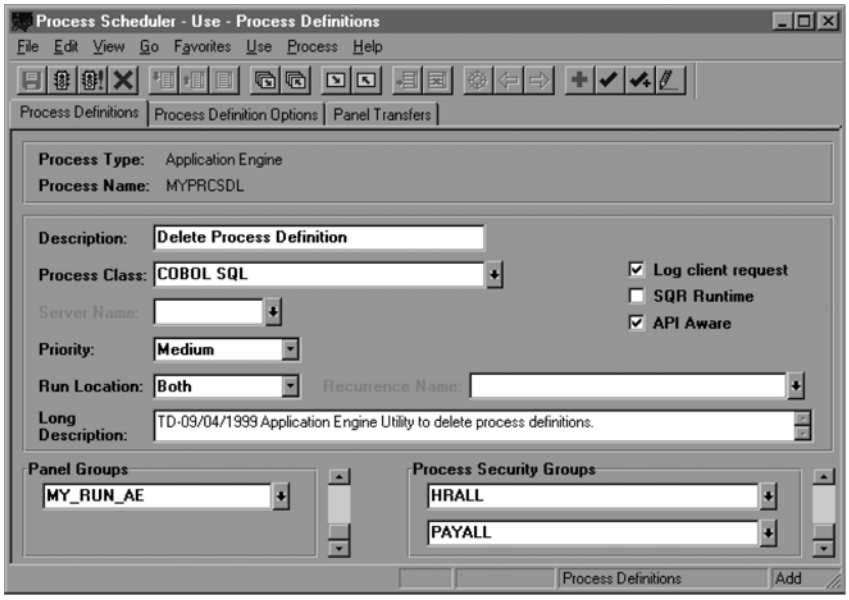


Figure 42.23 Adding process definition details

Our process definition is complete.

42.8.1 Create a DUMMY process definition for testing

While we're in Process Scheduler, let's create a dummy process definition that we'll use to test our application. We don't want to delete any existing process definitions. Let's add the dummy definition. We'll load random information since it's going to be deleted by our process.

Figure 42.24 shows a sample definition of a DUMMY process. We added a random panel group and process security groups. We'd like to see if our new application will delete process definition entries from a variety of tables. This panel will now contain DUMMY entries for the tables PRCSDEFN, PRCSDEFNPNL, and PRCSDEFNGRP.

We've also added some random panel transfer information for our DUMMY process definition. This will create a DUMMY entry in the table PRCSDEFNXFER.

Now, all that's left to do is create the actual Application Engine program!

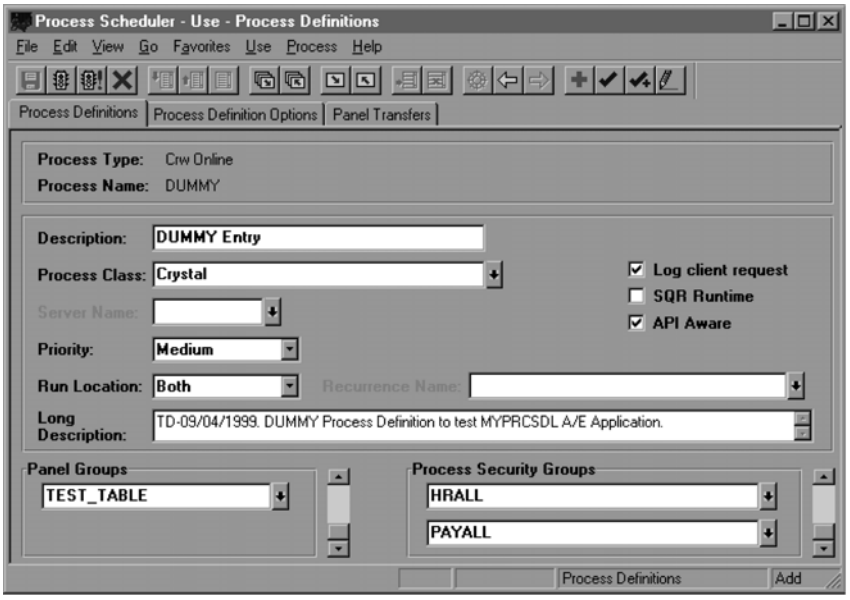


Figure 42.24 Adding a dummy process definition to test our application

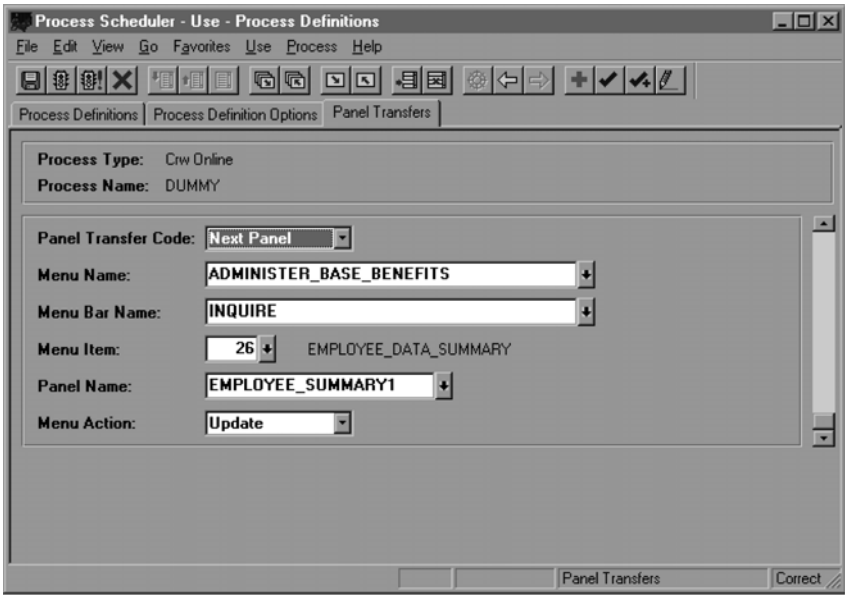


Figure 42.25 Adding a DUMMY process definition (Transfers)

KEY POINTS

1. The development life cycle for Application Engine programs is identical to that of SQR or COBOL development. The only difference is the Application Engine program itself.
2. When creating a Run Control record for Application Engine programs, use the AE_REQUEST record as a shell. Application Engine always uses the AE_REQUEST record so it's a good idea to integrate your new Run Control record with it. Use `RowInit` PeopleCode to assign your program name to the AE_PRODUCT and AE_APPL_ID fields in the AE_REQUEST record.
3. Also clone the AE_REQUEST panel when creating a new Run Control panel. The AE_REQUEST record will be the parent to your new Run Control record.
4. When creating the new panel group, add the AE_MESSAGE_LOG panel after your new Run Control panel This gives the user easy access to the message log entries for the completed run. The search record for your new panel group will be AE_REQUEST.

CHAPTER 43

Using Run Controls—part B

The Run Control panel for our new utility program is complete. We can access the new panel on the menu and even enter Run Control parameters. That's as far as we can go at the moment. If we click on the Traffic Light to initiate the process, an error will occur. That's because we haven't created the Application Engine program yet. This chapter will concentrate on the development of the Application Engine program. Once created, we can initiate the process through the Run Control panel. Some careful planning must be made to structure our program properly. Once complete, we can begin using our new tool to delete obsolete process definitions.

43.1 CREATE THE APPLICATION ENGINE PROGRAM

We're ready to begin developing our Application Engine program. It may be helpful to give a brief overview of our program structure:

```
MAIN.STEP1             Obtain Run Control Parameters
MAIN.STEP2             Display Run Control Parms on Message Log
MAIN.STEP3             Fetch Table One by One (DO Select)

   DYNSECTN.STEP1      Dynamically call PROCESS1 or PROCESS2

      PROCESS1.STEP1   Determine Number of Rows
      PROCESS1.STEP2   Process if exists (DO When > 0)
        DELETE1.STEP1  Delete Process Definition from table
      PROCESS1.STEP3   Call Message Routine
        MESSAGE.STEP1  Display Message

      PROCESS2.STEP1   Determine Number of Rows
      PROCESS2.STEP2   Process if exists (DO When > 0)
        DELETE2.STEP1  Delete Process Definition from table
      PROCESS2.STEP3   Call Message Routine
        MESSAGE.STEP1  Display Message
```

We need to make two key points. Consider the structure of our Application Engine program. The first key point has to do with the MAIN.STEP3 line. This step will select each of the six tables and process them one by one. Early in this chapter, we pointed out that four of the process definition tables are prefixed with "PS_", and two are not. We'll use two separate processes (PROCESS1 and PROCESS2) to handle both types. They will be called dynamically based on the table being processed. The step DYNSECTN.STEP1 will call either PROCESS1 or PROCESS2.

The second point has to do with the processing steps we have chosen. Some readers may think we have taken the long way in performing our task. This may be true depending on the database you are using. For instance, if you're an Oracle user, you certainly don't need to determine if the row exists before deleting it. This is not true for all databases though. In DB2, you may receive an error if you try to delete a row that doesn't exist. We would then need to add additional error-handling steps. We could also use the DB platform field to code individual routines based on your particular database. This would hardly seem practical for the purposes of this book. Let's move on now.

We'll begin developing our program from the minor routines on up. Since we've established the program hierarchy, we can work backward and not worry about step dependencies we would encounter by going forward.

43.1.1 Building the MESSAGE section

We'll create the section MESSAGE first.

Navigation: Go →PeopleTools →Application Engine →Use →Application Engine → Application →Add

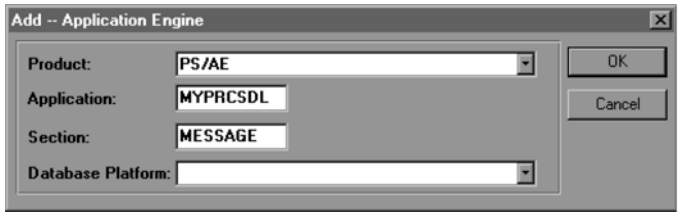


Figure 43.1 Creating the MESSAGE section

First tab over to the application definition (figure 43.2). Fill in the description, cache record, version, and message set number. We'll be using our updated cache record (USER_AET) and message set from prior exercises. Also, set the trace parameter to SQL. After we test our application, we'll examine the trace file.

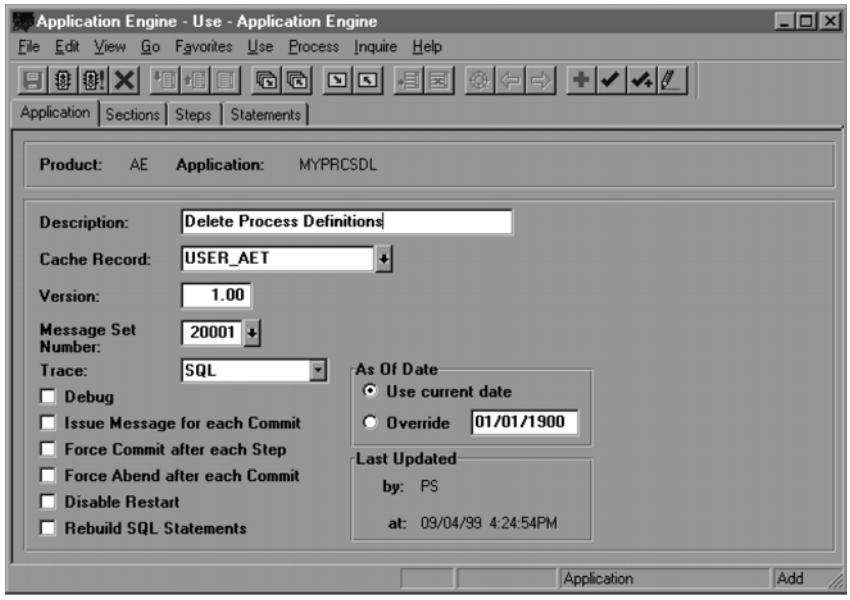


Figure 43.2 Defining our application MYPRCSDL

Fill in the description of the MESSAGE section (figure 43.3). This section simply writes a message to the message log.

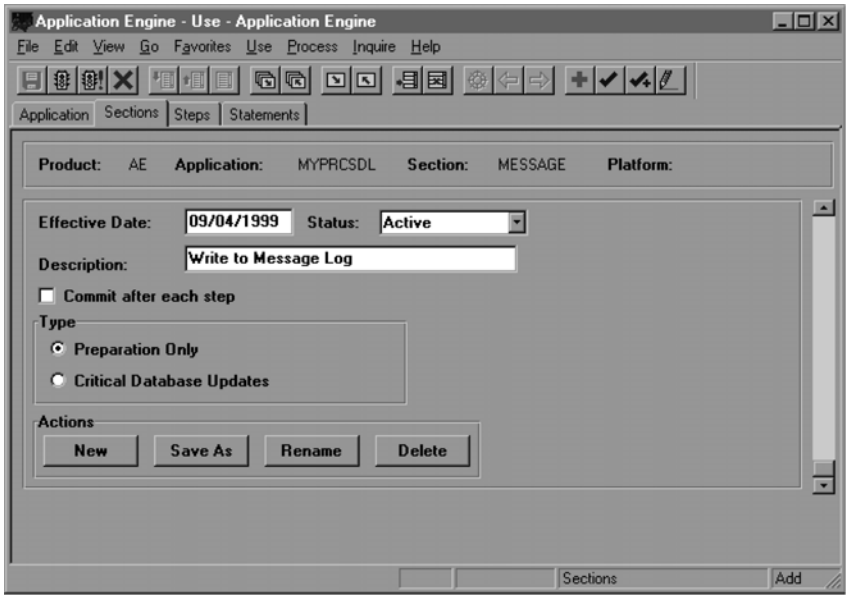


Figure 43.3 Defining the MESSAGE section

The only parameter you need to fill in for the first (and only) step of the MESSAGE section is the step name. We call it STEP1 (figure 43.4).

Figure 43.5 shows the message statement we use. This is similar to prior exercises. We pass the record name (RECNAME) and number of rows (COUNTER) to the message log.

We're done with the MESSAGE section.

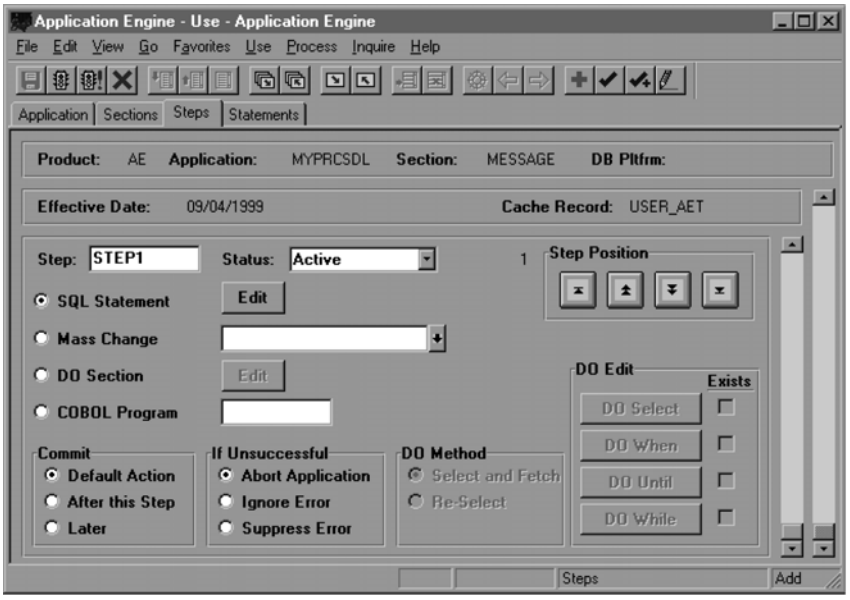


Figure 43.4 Defining STEP1 of the MESSAGE section

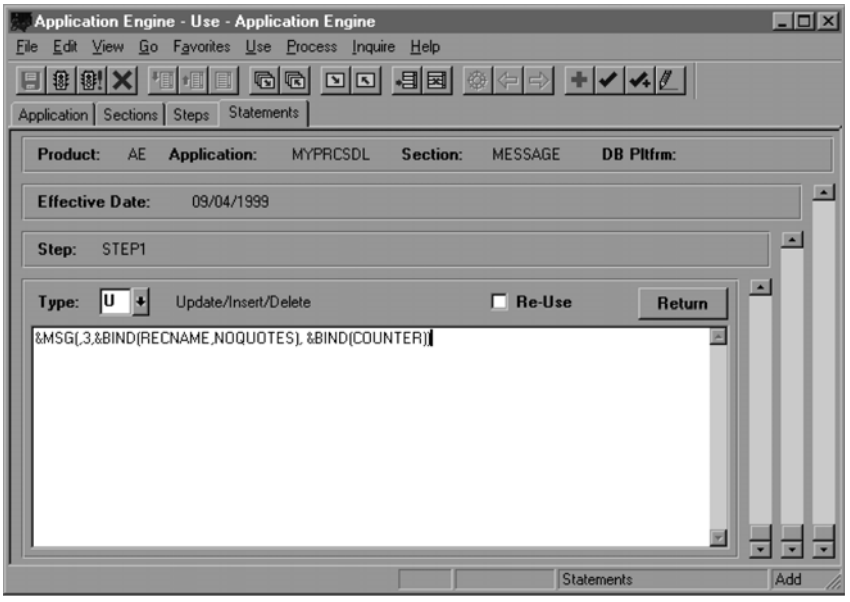


Figure 43.5 Adding our message statement

43.1.2 Building the DELETE1 section

We now add the DELETE1 section (figure 43.6), which performs the `Delete` against the process definition tables that have the standard 'PS_' prefix.

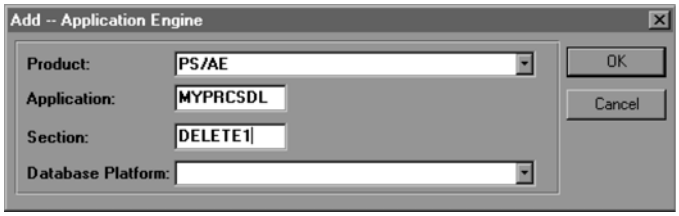


Figure 43.6 Adding the DELETE1 section

The only thing we need to add is the description. `'Delete Table with PS_'` is a fairly accurate description (figure 43.7).

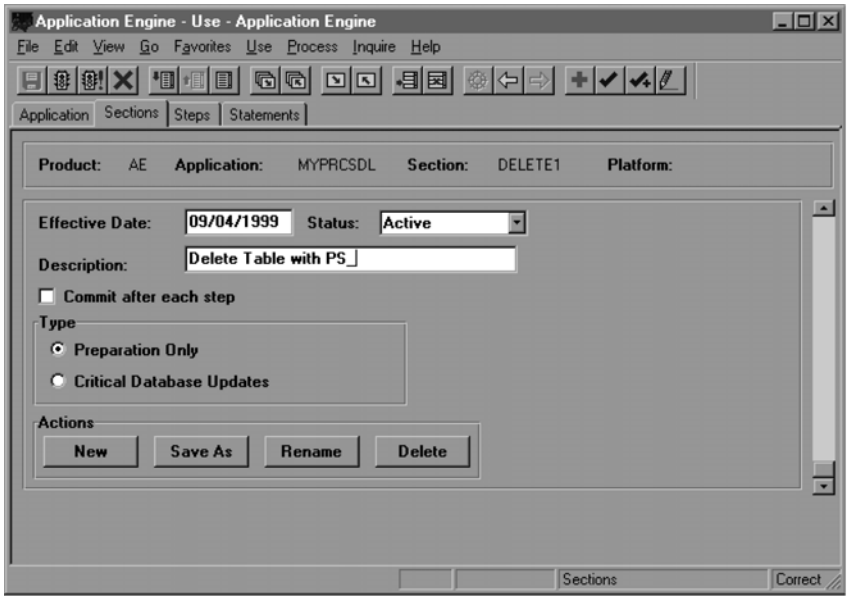


Figure 43.7 Defining the DELETE1 section

We call this first (and only) step of the DELETE1 section STEP1 (figure 43.8).

We've added the SQL statement to `Delete` a row from the table specified by the RECNAME cache field (figure 43.9). Notice the table name is in the same dynamic format used in prior exercises. The DELETE1 section will be part of the process that handles records with the PS_ prefix.

The DELETE1 section is complete.

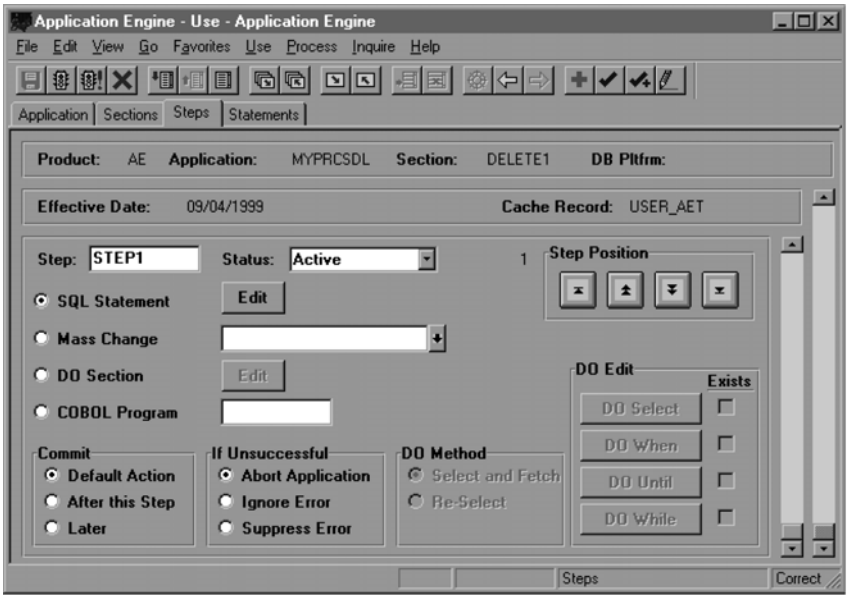


Figure 43.8 Defining STEP1 of the DELETE1 section

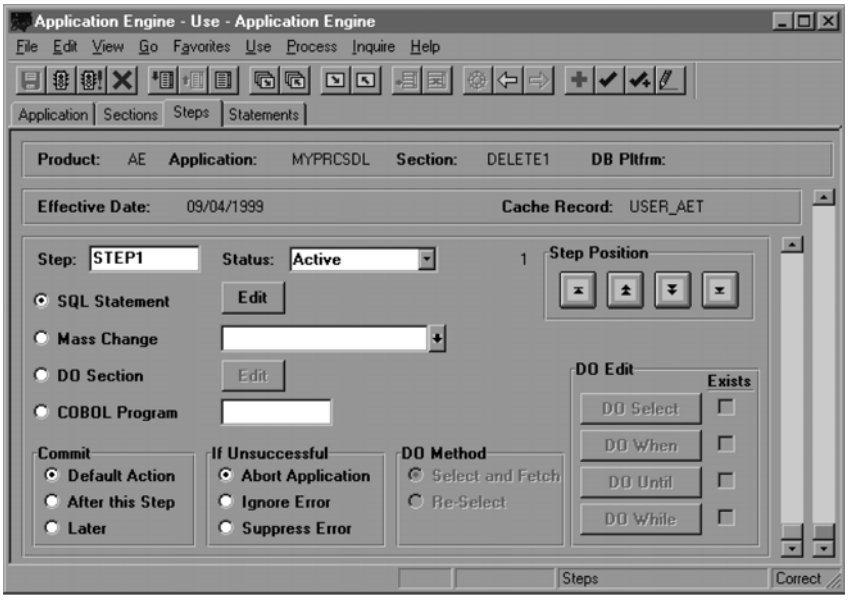


Figure 43.9 Defining STEP1 of the DELETE1 section

43.1.3 Building the DELETE2 section

We add the DELETE2 section (figure 43.10). This section performs the `Delete` against the process definition tables that DO NOT utilize the standard PS_ prefix.

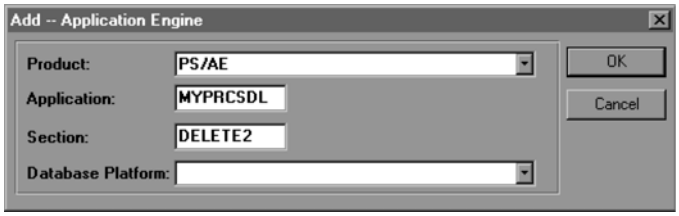


Figure 43.10 Adding the DELETE2 section

Once again, the only thing we need to add is the description. `'Delete Table without PS_'` is perfect for our section (figure 43.11).

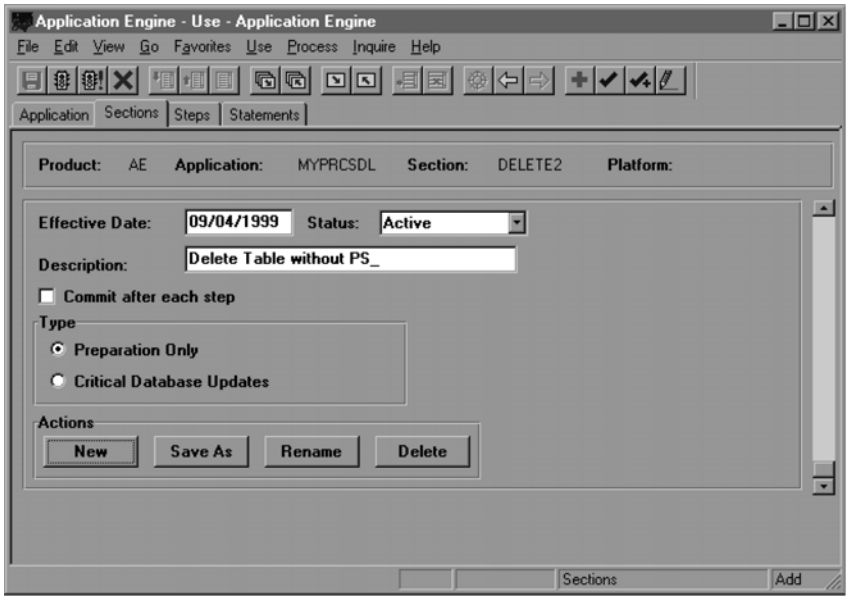


Figure 43.11 Defining the DELETE2 section

TIP Existing sections can be "cloned" using the Save As button on the Section Definition panel. We can then modify the new section as needed. In the case of the DELETE1 and DELETE2 sections, the modifications are minimal. For purposes of these exercises, we'll create each section manually.

Let's call this first (and only) step of the DELETE2 section STEP1 (figure 43.12).

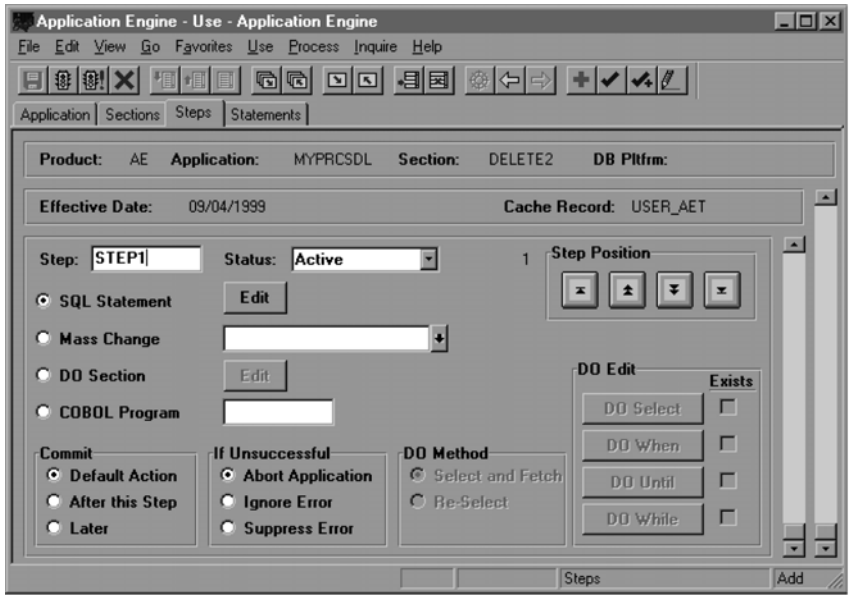


Figure 43.12 Defining STEP1 of the DELETE2 section

The delete statement in figure 43.13 is almost identical to the one in the DELETE1 section. The only difference is the absence of the PS_ prefix.

The DELETE2 section is complete.

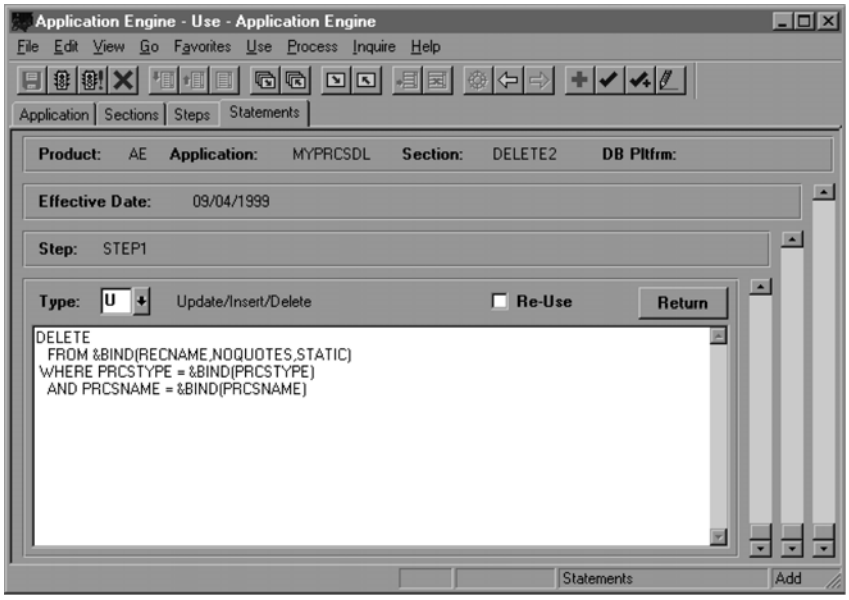


Figure 43.13 The Delete statement for DELETE2.STEP1

43.1.4 Building the PROCESS1 section

Next, let's develop the PROCESS1 section. This section will handle all process definition tables that require the PS_ prefixed to the RECNAME. Add the PROCESS1 section now (figure 43.14).

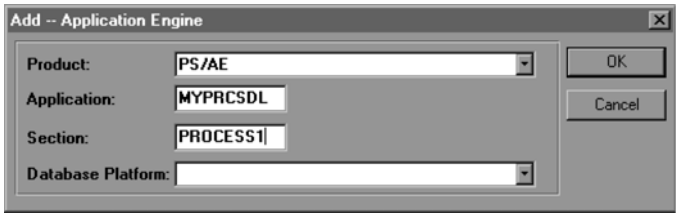


Figure 43.14 Adding the PROCESS1 section

Fill in the description for the PROCESS1 section. `'Process Table with PS_'` is the description we'll use for our section (figure 43.15).

Our first step in this section determines the number of rows in the particular process definition table, which is stored in the RECNAME cache field. The only parameter we need to enter in the Step Definition panel is the name of our step. We'll call it STEP1 (figure 43.16).

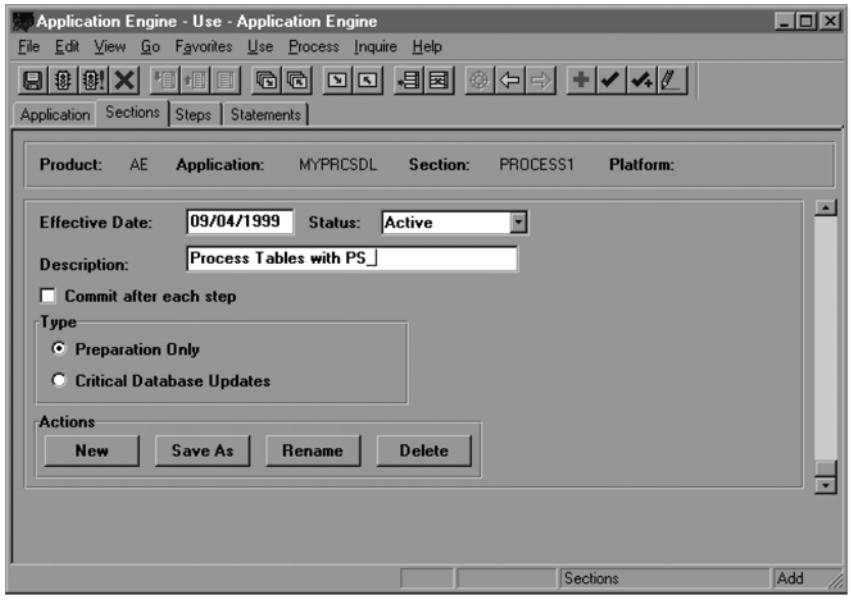


Figure 43.15 Defining the PROCESS1 section

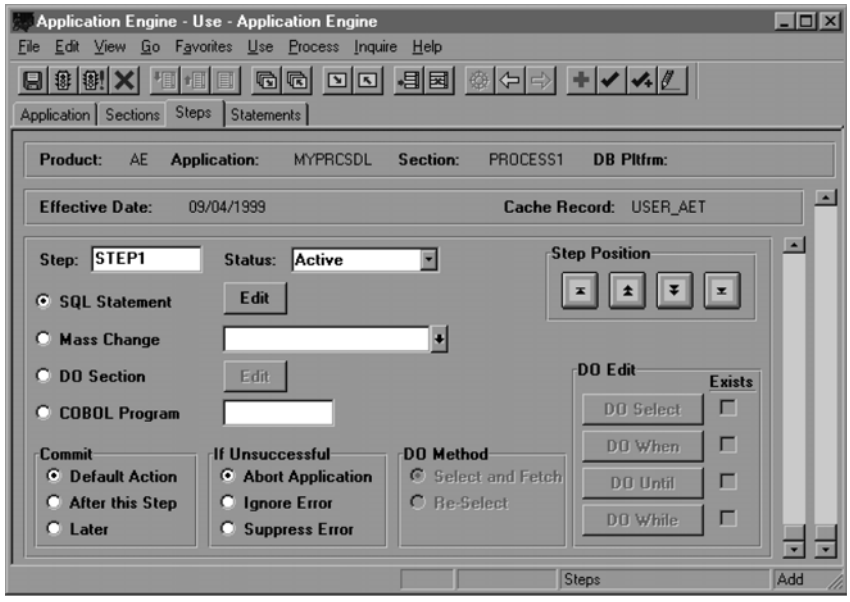


Figure 43.16 Adding STEP1 to the PROCESS1 section

We add a simple `Select` statement to retrieve the number of rows in the table (specified in RECNAME) and populate the COUNTER cache field.

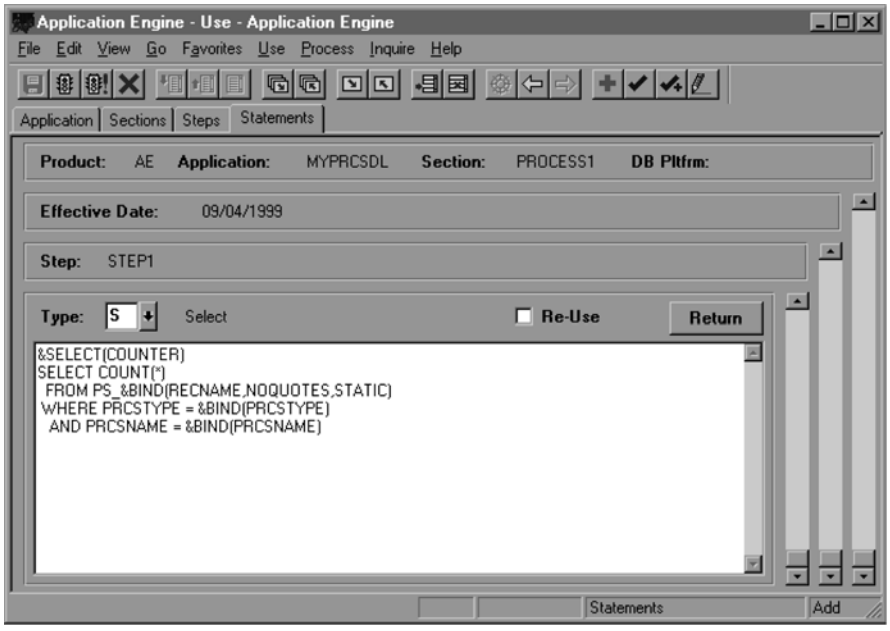


Figure 43.17 Adding the Select statement to PROCESS1.STEP1

Let's add another step to the PROCESS1 section. Click on the Steps Folder tab to return to the Step Definition panel. Place the cursor in the Step field and press the F7 key to insert a new row. Our next step will be named STEP2 (figure 43.18). Click on the DO section radio button then click on the corresponding edit button. Add the DELETE1 section when the DO section dialog box appears. You can see the DELETE1 section name next to the DO section edit button when you return. Our next step is to populate the statement panel with a `DO When` statement. The DELETE1 section is performed only if there are rows in the table containing the process definition from the Run Control record. Let's add the `DO When` statement now.

This is the same statement we've used in previous exercises. If the COUNTER cache field contains a value greater than zero, a `True` condition is returned, and the section DELETE1 is performed. If the COUNTER cache field contains a value of zero, a `False` condition is returned, and the DELETE1 section is not performed.

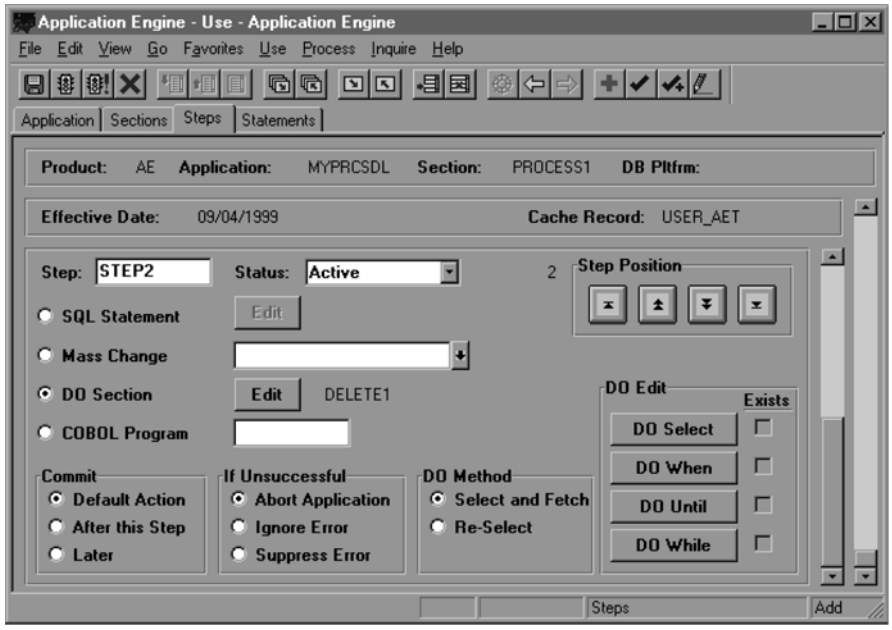


Figure 43.18 Adding STEP2 to the PROCESS1 section

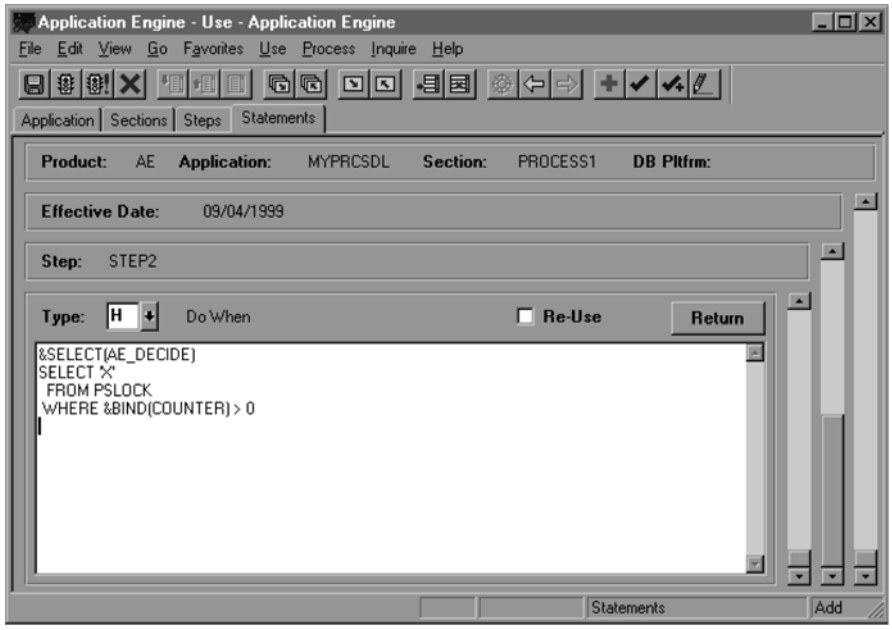


Figure 43.19 Adding the DO When statement to PROCESS1.STEP2

Use the F7 key again to insert a new step in the PROCESS1 section. Let's call it STEP3 (figure 43.20). Its function is to call the MESSAGE section we've created. Use the DO section radio button and edit box to set the section to MESSAGE. The MESSAGE section simply writes a message log entry containing the record name and number of rows processed.

The PROCESS1 section is complete.

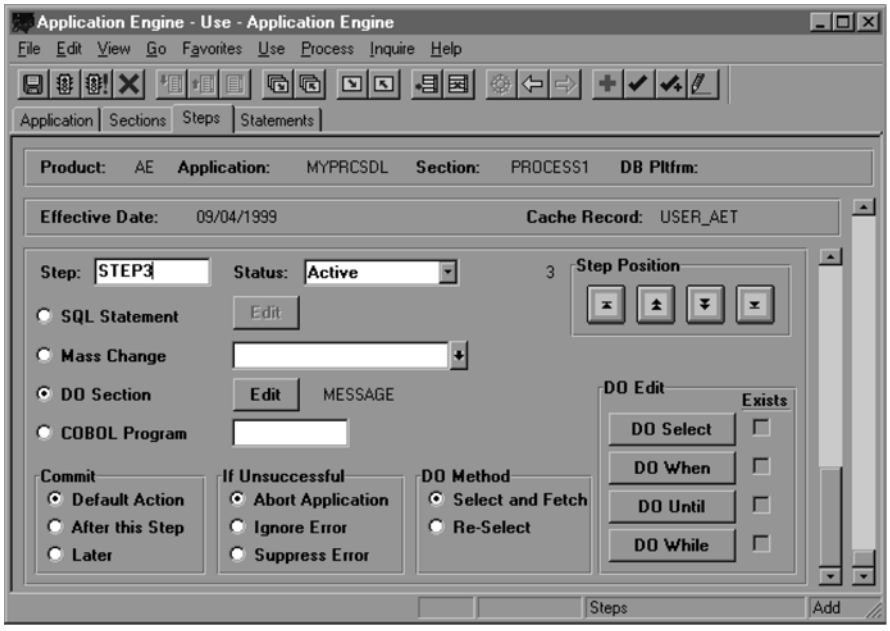


Figure 43.20 Adding STEP3 to the PROCESS1 section

43.1.5 Building the PROCESS2 section

Next, let's develop the PROCESS2 section. This section handles all process definition tables that do not require the PS_ prefix to the RECNAME. The PROCESS2 section is almost identical to the PROCESS1 section. Add the PROCESS2 section now (figure 43.21).

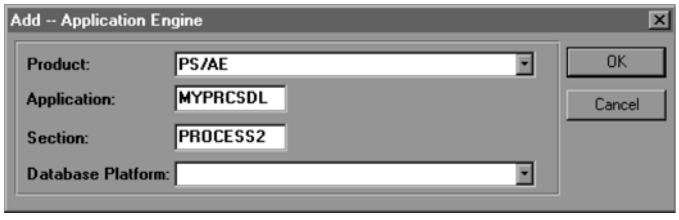


Figure 43.21 Adding the PROCESS2 section

Fill in the description for the PROCESS2 section. `'Process Table without PS_'` is the description we'll use for our section (figure 43.22).

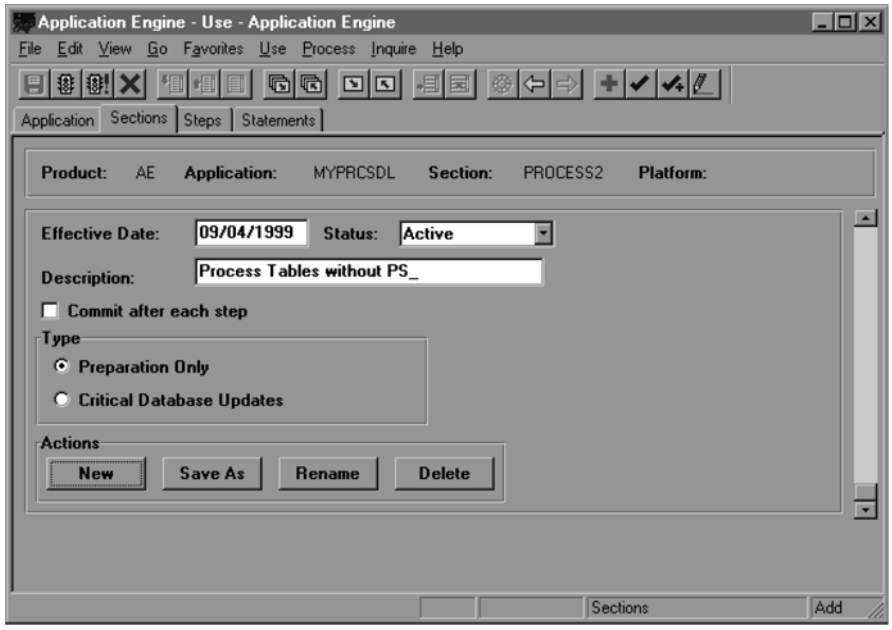


Figure 43.22 Defining the PROCESS2 Section

TIP You can try cloning the PROCESS1 section to produce the new section PROCESS2 using the 'Save As' button. Make the alterations to the statements as you would if you had created the PROCESS2 section manually.

Our first step in this section determines the number of rows in the particular process definition table stored in the RECNAME cache field. The only parameter we need to enter in the Step Definition panel is the name of our step. We'll call it STEP1 (figure 43.23).

Next, we add a simple `Select` statement to retrieve the number of rows in the table (specified in RECNAME) and populate the COUNTER cache field. It is nearly identical to the `Select` statement found in STEP1 of the PROCESS1 section, the difference being the absence of the PS_ prefix preceding the table name.

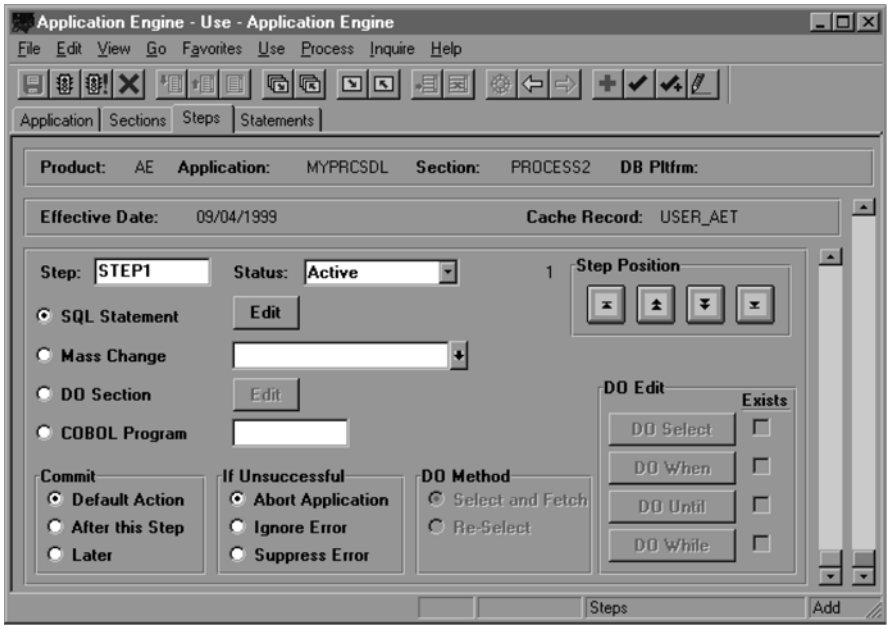


Figure 43.23 Adding STEP1 to the PROCESS2 section

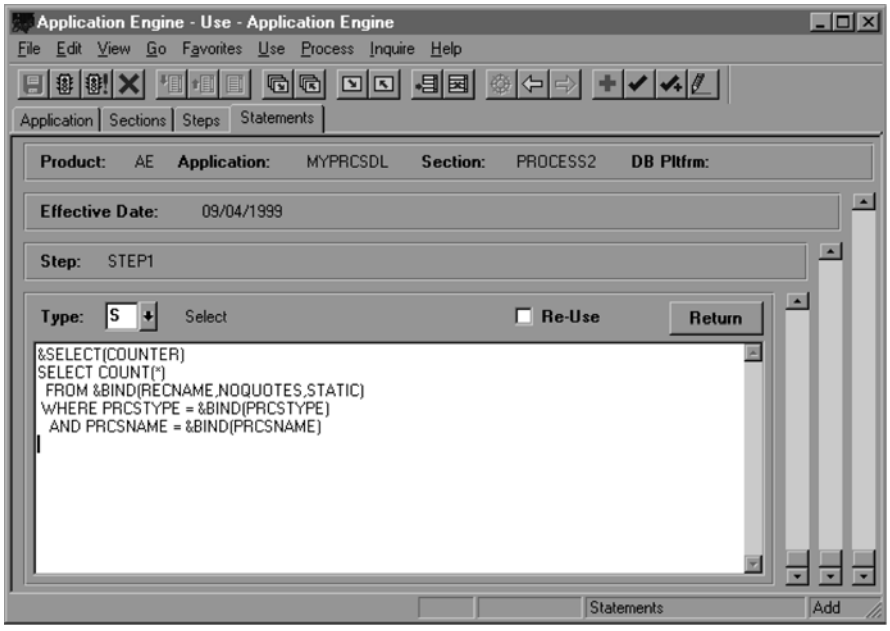


Figure 43.24 Adding the Select statement to PROCESS2.STEP1

Let's add another step to the PROCESS2 section. Click on the Step Folder tab to return to the Step Definition panel. Place the cursor in the Step field and press the F7 key to insert a new row. Our next step is named STEP2 (figure 43.25). Click on the DO section radio button, then on the corresponding edit button. Add the DELETE2 section when the DO section dialog box appears. You can see the DELETE2 section name next to the DO section edit button when you return. Our next step is to populate the statement panel with a `DO When` statement. The DELETE2 section is performed only if there are rows in the table containing the process definition from the Run Control record. Let's add the `DO When` statement now.

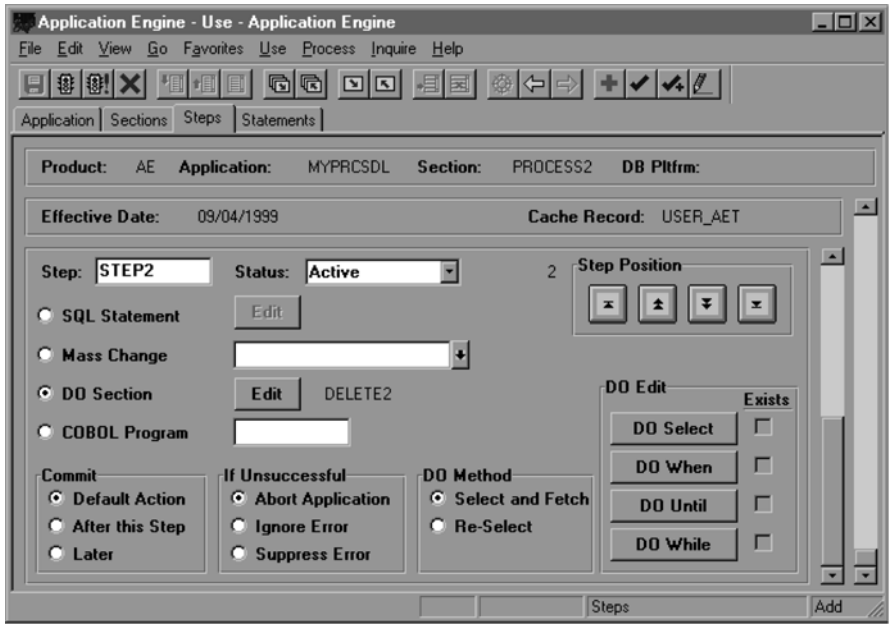


Figure 43.25 Adding STEP2 to the PROCESS2 section

Our DO When statement (figure 43.26) is the same as before. If the COUNTER cache field contains a value greater than zero, a `True` condition is returned, and the section DELETE1 is performed. IF the COUNTER cache field contains a value of zero, a `False` condition is returned, and the DELETE2 section is not performed.

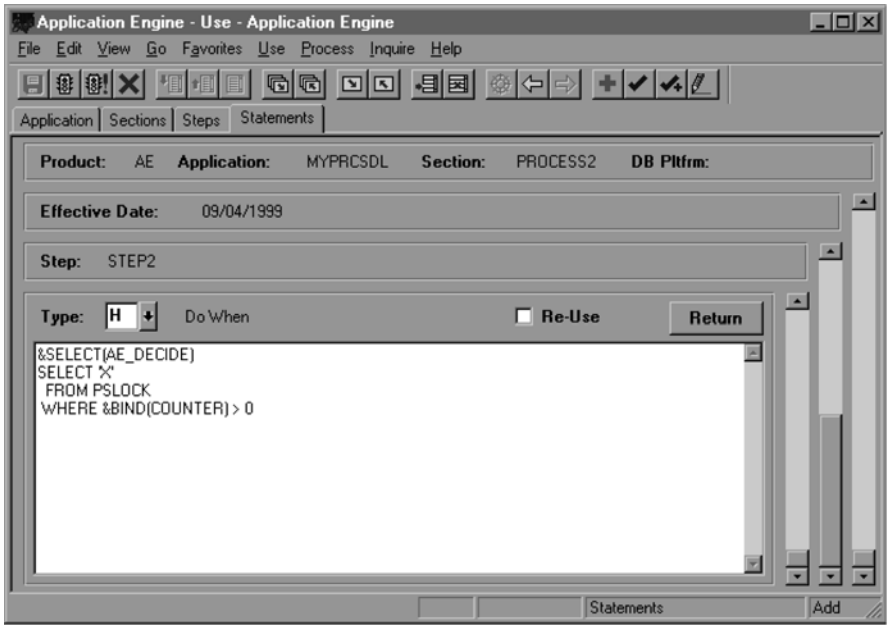


Figure 43.26 Adding the DO When statement to PROCESS2.STEP2

Use the F7 key again to insert a new step in the PROCESS2 section. Let's call it STEP3 (figure 43.27). Its function is to call the MESSAGE section we've created. Use the DO section radio button and edit box to set the section to MESSAGE. As stated in the creation of STEP3 in the PROCESS1 section, the MESSAGE section simply writes a message log entry containing the record name and number of rows processed.

The PROCESS2 section is complete.

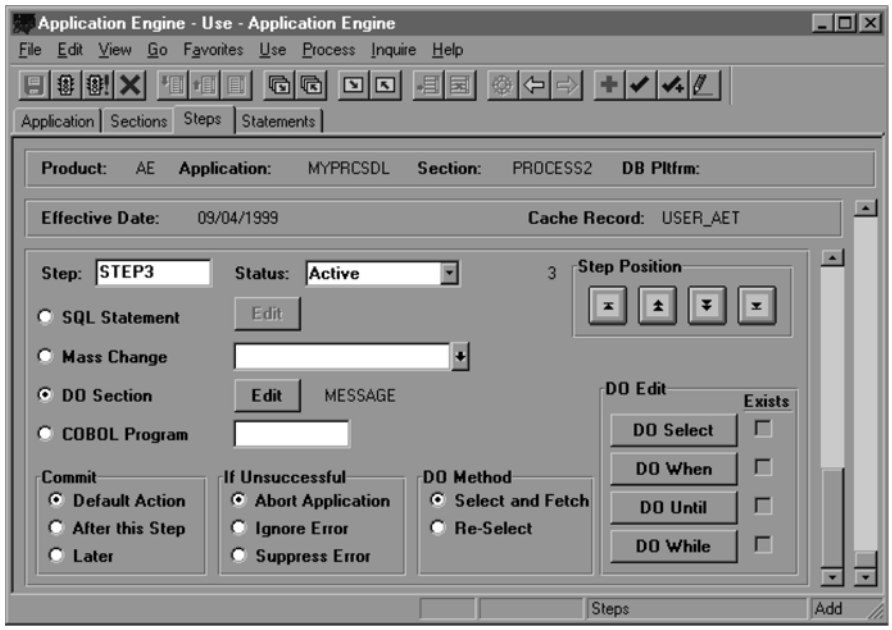


Figure 43.27 Adding STEP3 to the PROCESS2 section

43.1.6 Building the DYNSECTN section

Add a new section called DYNSECTN (figure 43.28). The purpose of this routine is to dynamically call either PROCESS1 or PROCESS2.

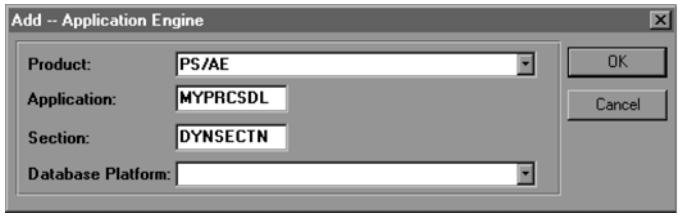


Figure 43.28 Adding the DYNSECTN section

First, add the description for the DYNSECTN section (figure 43.29).

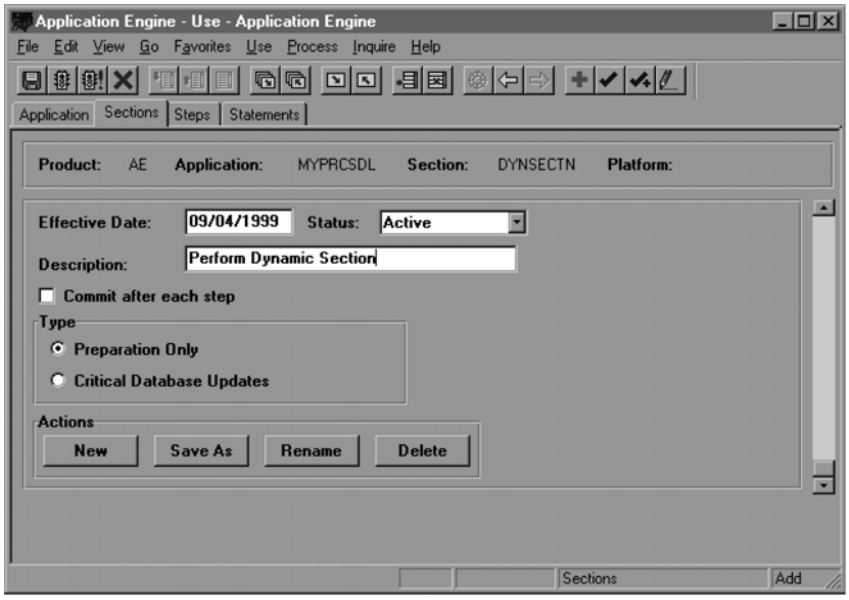


Figure 43.29 Defining the DYNSECTN section

On the Step Definition panel fill in the step name with STEP1 (figure 43.30).

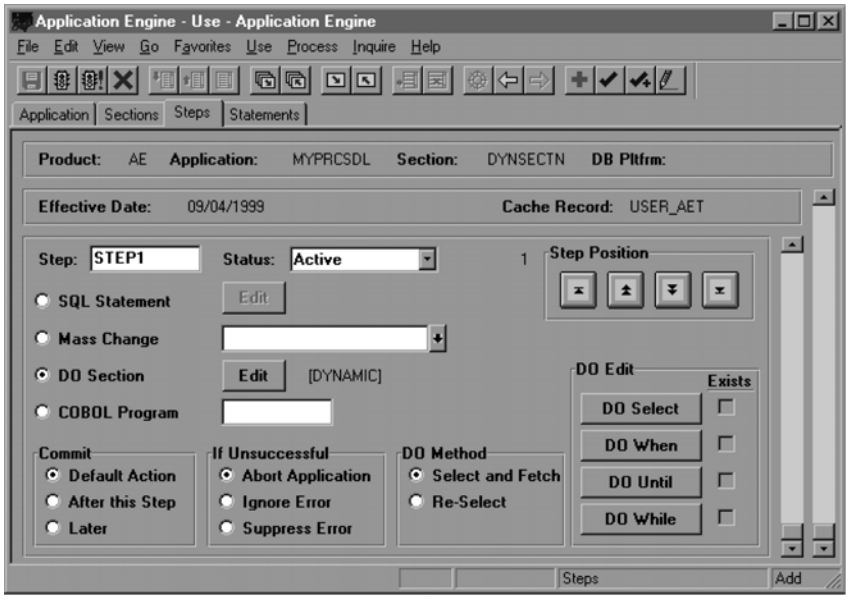


Figure 43.30 Adding STEP1 to the DYNSECTN section

Next, click on the DO section radio button and press the corresponding edit button. When the DO section dialog box appears, click on "Dynamic Section." Notice the literal DYNAMIC (in brackets) next to the DO section edit button (discussed in chapter 41). The symbolic parameter `&SECTION` is placed in the AE_DO_SECTION column in the AE_STEP_TBL. This is the table that's populated by your Step Definition panel entries. When the step is executed, the value of the AE_SECTION cache field will be used in place of the `&SECTION` symbolic. Our next and final section, MAIN, populates the AE_SECTION cache field with the value PROCESS1 or PROCESS2. The DYNSECTN section then performs PROCESS1 or PROCESS2.

The DYNSECTN section is complete.

43.1.7 Building the MAIN section

As we've discussed, all Application Engine programs must begin with a section called MAIN. Let's add the MAIN section now (figure 43.31).

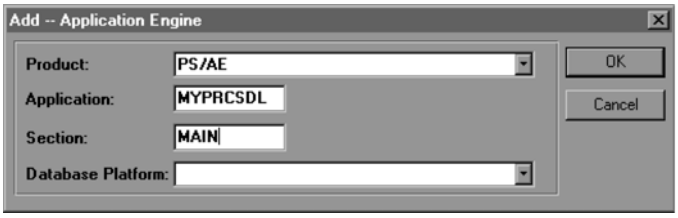


Figure 43.31
Adding the MAIN section

The Section Definition panel is displayed in figure 43.32. We added a simple description of the MAIN section: `Delete Process Definitions`.

We'll call the first step of the MAIN section STEP1 (figure 43.33). The purpose of this step is to obtain the processing parameters from our Run Control record.

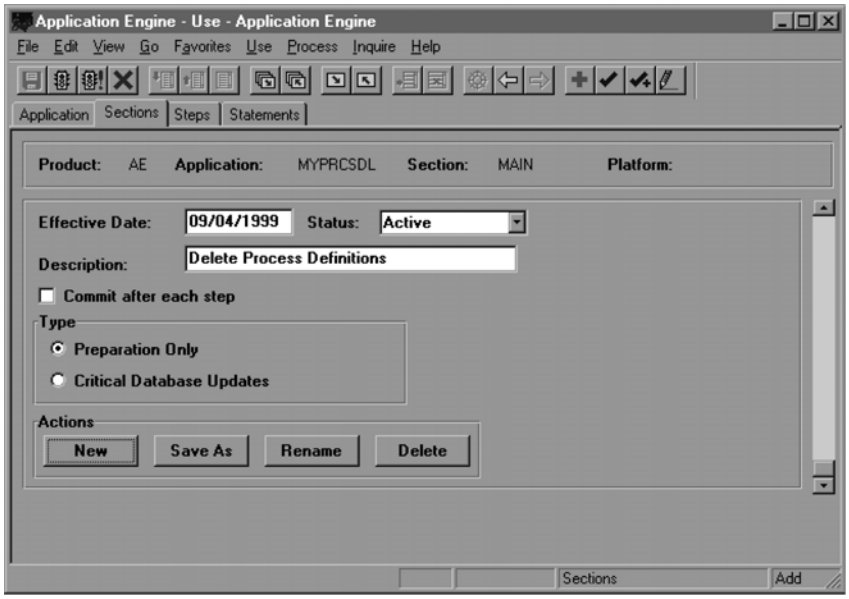


Figure 43.32 Defining the MAIN section

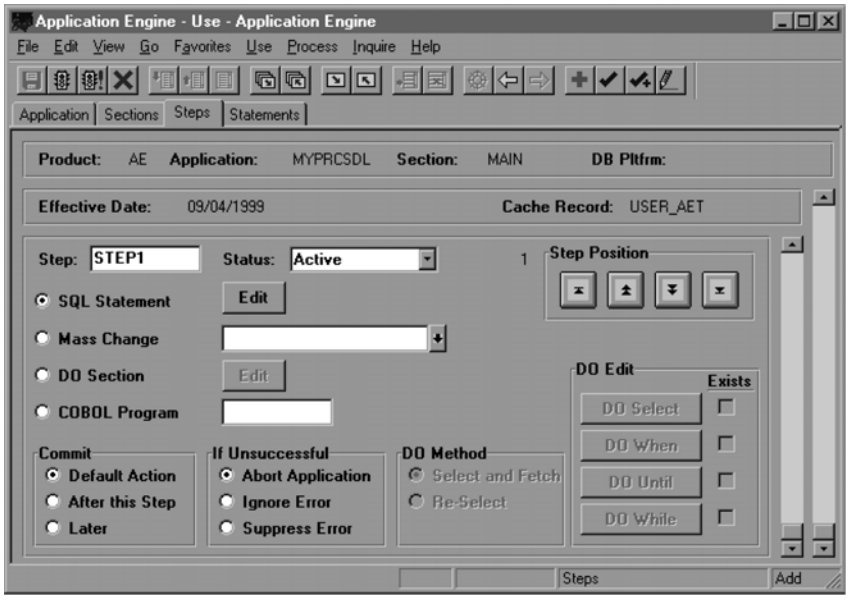


Figure 43.33 Adding STEP1 To the MAIN section

Figure 43.34 shows the `Select` statement for MAIN.STEP1. To retrieve our run parameters, we need to join our Run Control record (MY_RUN_CNTL_AE) to the Application Engine Run Control record called AE_RUN_CONTROL. The AE_RUN_CONTROL record discussed earlier in the book is used by Application Engine to hold run information such as PROCESS_INSTANCE, OPRID, RUN_CNTL_ID, and REQUEST_NBR. It also stores information about each run such as the last step committed, used when restarting an Application Engine program that may have terminated due to errors.

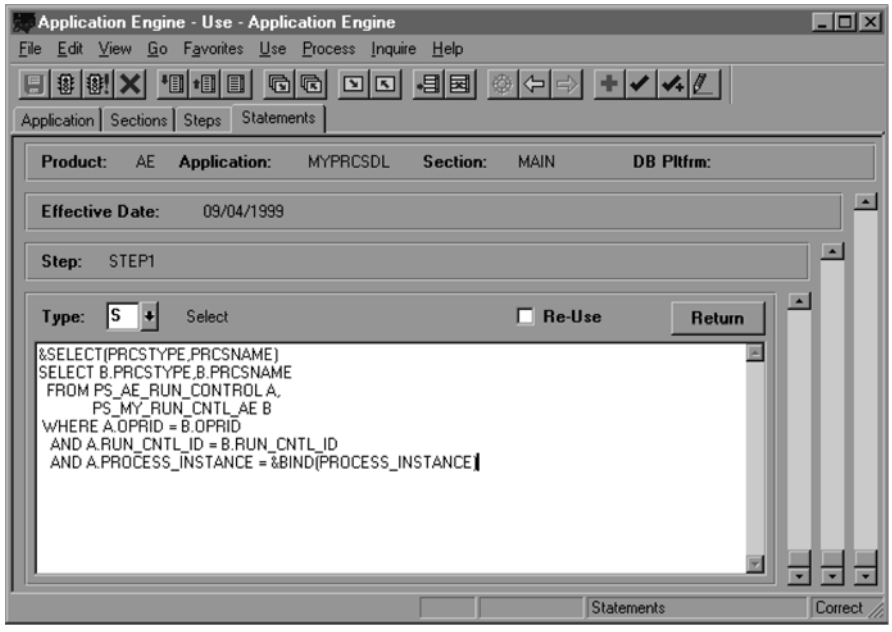


Figure 43.34 Select statement to retrieve our Run Control parameters

The records MY_RUN_CNTL_AE and AE_RUN_CONTROL are joined by the OPRID and RUN_CNTL_ID fields. The AE_RUN_CONTROL record is selected for the appropriate PROCESS_INSTANCE assigned by the Process Scheduler. The `&SELECT` statement stores the `PRCSTYPE` and `PRCSNAME` Run Control parameters in our cache record. We now have access to the process parameters entered on the Run Control panel. We'll display them on the message log in our next step.

Return to the Step Definition panel by clicking the Steps Folder tab. Place the cursor in the step field and press F7 to insert a new row. We'll call the new step STEP2 (figure 43.35). Now, click on the Statements Folder tab.

We write a simple message to the message log displaying the process type (`PRCSTYPE`) and process name (`PRCSNAME`) taken from the Run Control record. You

can see the `&MSG` syntax in figure 43.36. Remember, the `&MSG` function only works with a statement Type of "`U`" (`Update/Insert/Delete`).

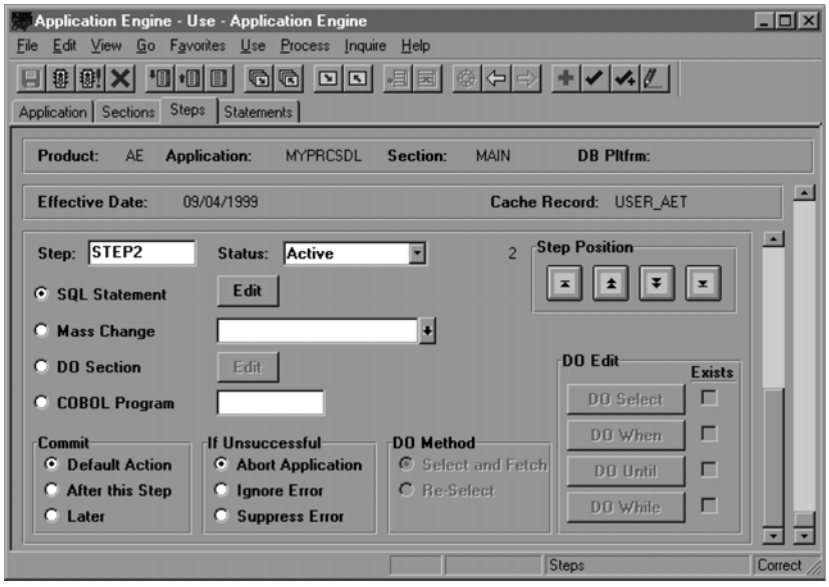


Figure 43.35 Adding STEP2 to the MAIN section

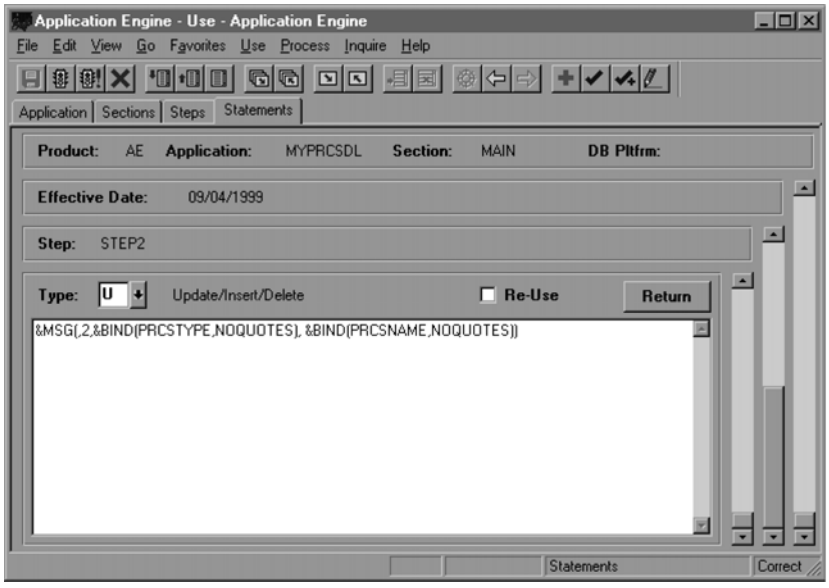


Figure 43.36 Run Control parameters are written to the message log

On the Step Definition panel, press F7 to insert another step. We'll call it STEP3 (figure 43.37). Click on the DO section radio button and the corresponding edit button. When the DO section dialog box appears, enter the section DYNSECTN. This is the section that handles the dynamic call of either PROCESS1 or PROCESS2. Let's code the `DO Select` statement for STEP3 in our MAIN section.

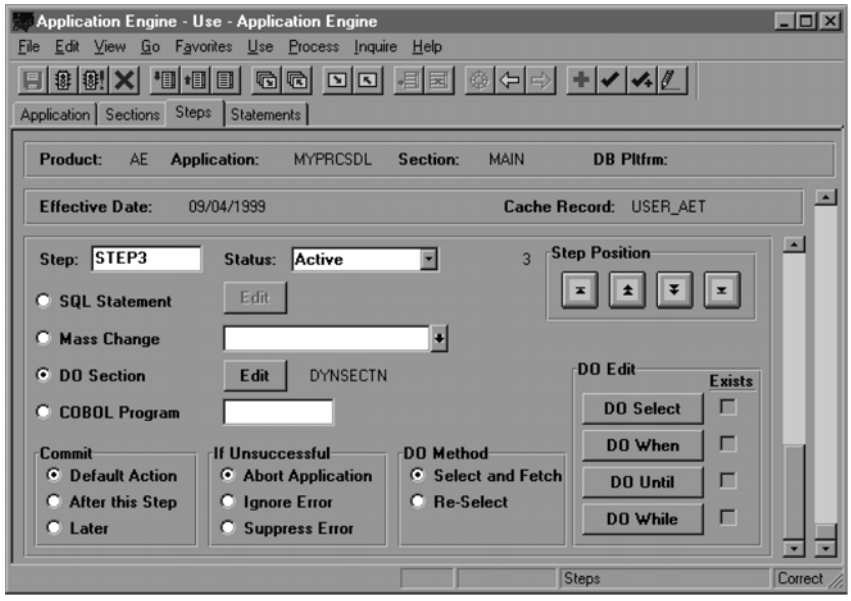


Figure 43.37 Adding STEP3 to the MAIN section

Figure 43.38 shows the `DO Select` statement that controls the processing of the six Process Definition tables. Let's take a closer look at the statement and describe what's happening:

```
&SELECT(AE_SECTION,RECNAME)
SELECT 'PROCESS1', RECNAME
  FROM PSRECDEFN
 WHERE RECNAME = 'PRCSDEFN'
    OR RECNAME = 'PRCSDEFNGRP'
    OR RECNAME = 'PRCSDEFNPNL'
    OR RECNAME = 'PRCSDEFNXFER'

UNION

SELECT 'PROCESS2', RECNAME
  FROM PSRECDEFN
 WHERE RECNAME = 'PSPRCSRQST'
    OR RECNAME = 'PSPNLFIELD'
 ORDER BY 1,2
```

Assign cache field values

Select process definition tables that require the PS_ prefix. Each row selected returns two columns – 'PROCESS1' and RECNAME. 'PROCESS1' populates the AE_SECTION cache field.

Using a UNION, select process definition tables that do not use the PS_ prefix. The rows returned will have the columns 'PROCESS2' and RECNAME. 'PROCESS2' will be stored in the AE_SECTION cache field.

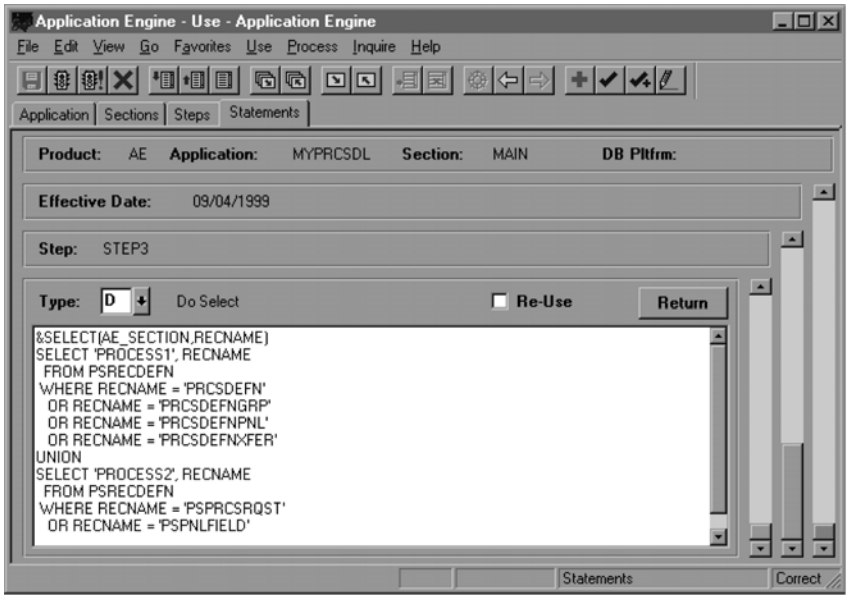


Figure 43.38 The DO Select statement to process the six process definition tables

The SQL `DO Select` statement above may appear strange at first. Let's talk about what we're trying to accomplish. Our goal is to delete any process definitions that match the process type and process name entered on the Run Control panel. There are six process definition tables that may contain the process definition entered. We're going to produce a result set that contains the record name (RECNAME) for each of the six tables along with the name of the dynamic section to use. Each row returned will then be processed by the appropriate section. PROCESS1 handles the tables with the PS_ prefix, and PROCESS2 handles the tables without the PS_ prefix. Let's test our statement using a database tool outside of PeopleSoft, SQL*Talk in this example.

Figure 43.39 shows the `Select` statement results using SQL*Talk. The first four tables use the PROCESS1 section while the last two tables use the PROCESS2 section. For each of these rows, the cache fields AE_SECTION and RECNAME are updated. The section DYNSECTN is performed for each row, which in turn calls either the PROCESS1 or PROCESS2 sections dynamically, based on the contents of the AE_SECTION cache field.

Our Application Engine program is finally complete! We can test our new utility to delete obsolete process definitions. It's perfectly normal to feel a bit nervous or excited before testing your work. We've put considerable effort into this application, and now we'll see if it has paid off. Let's begin our test.

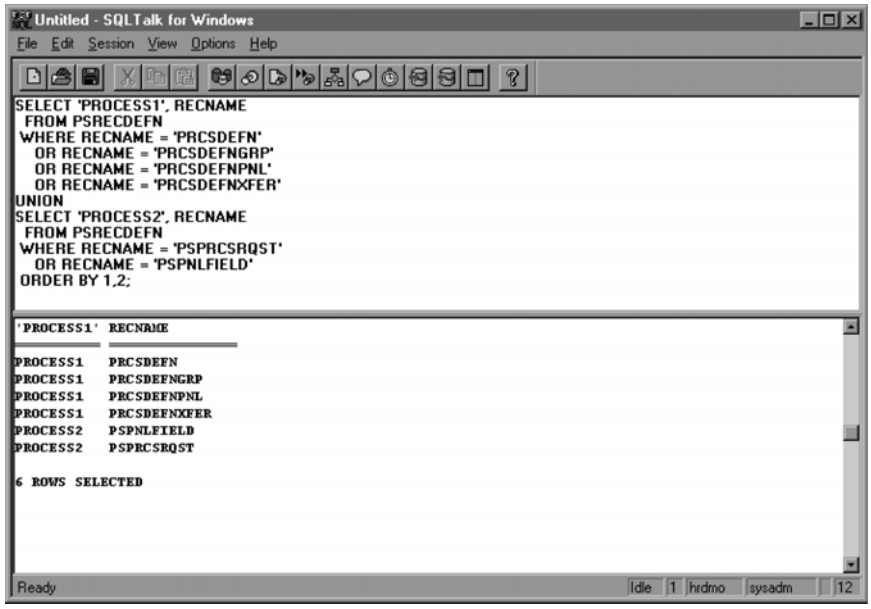


Figure 43.39 Testing our statement using SQL*Talk

43.2 TESTING THE COMPLETED APPLICATION

Figure 43.40 shows the navigation to our new utility panel. Add a new Run Control ID. I'm going to use a Run Control ID called "DUMMY" since we're going to delete the DUMMY process definition we created earlier in this chapter.

Navigation: Go →PeopleTools →Utilities →Process →Delete Process Definition → Delete PRCSDEFN →Add

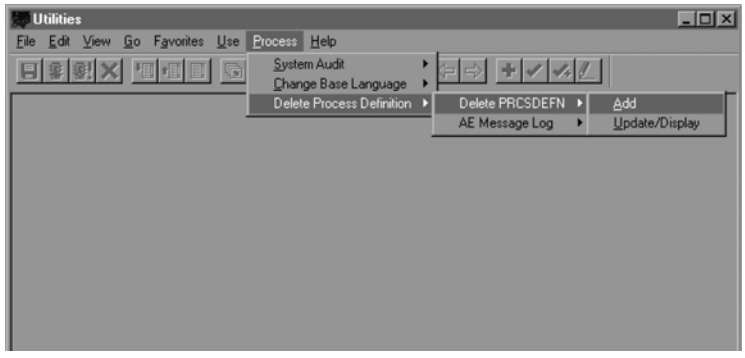


Figure 43.40 Accessing the Delete Process Definition panel

We've entered the process type and process name used for our DUMMY definition (figure 43.41). Remember, we created a DUMMY definition to test our process. We don't want to delete a "real" process definition.

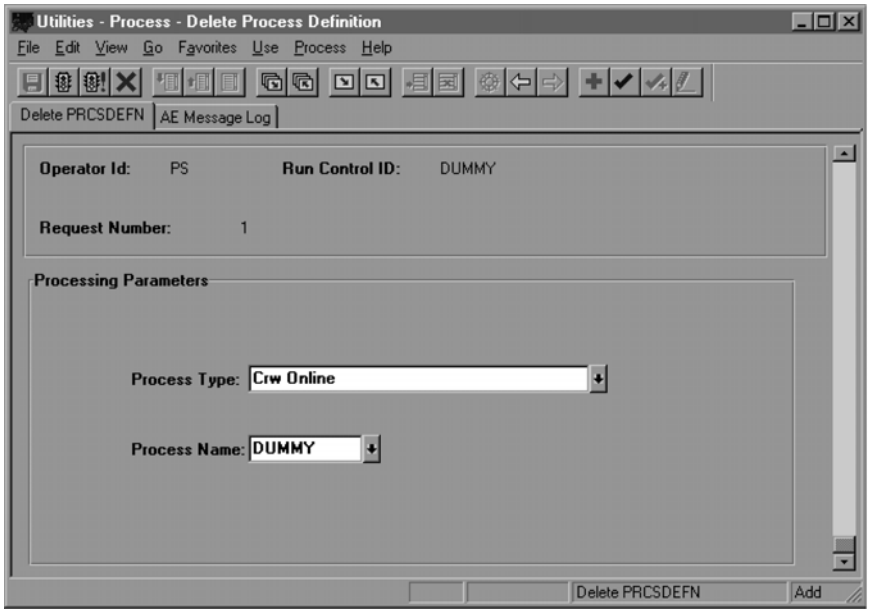


Figure 43.41 Assigning parameter values on the Run Control panel

Save the record and click on the Traffic Light to initiate a Process Scheduler request.

Figure 43.42 shows the Process Request panel. Our Application Engine program (MYPRCSDL) appears in the panel. Our process definition for MYPRCSDL appears to be functioning correctly, so click OK to initiate the process within Process Scheduler.

This is a good sign! You may notice the MS-DOS box appears when running on the client (figure 43.43). The lines displayed may move very quickly on the screen. Figure 43.43 shows some of the lines displayed in the MS-DOS box. Near the top of the screen, I can see our processing parameters in the Run Control record. Near the bottom, we see the first record (PRCSDEFN) had one row with our DUMMY process definition. Also notice the dynamic section PROCESS1 was performed as planned.

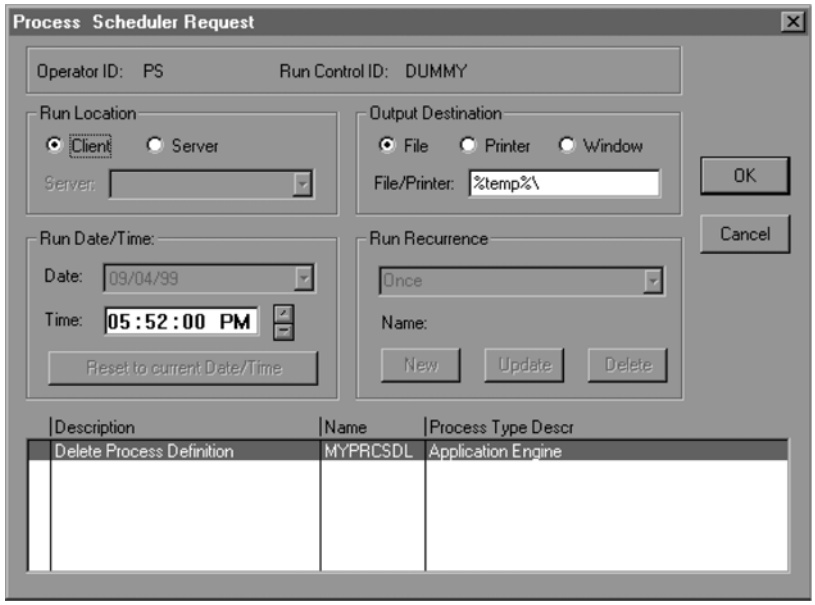


Figure 43.42 Submitting the Process Scheduler request

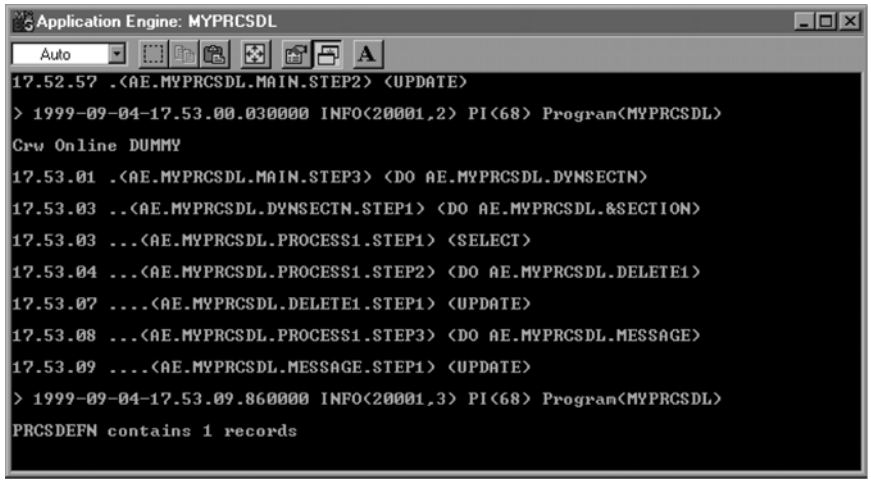


Figure 43.43 The MS-DOS box appears for our process

We still need to verify that the process functioned correctly, especially when a `Delete` statement is being executed. When the process has completed, we'll click on the A/E Message Log Folder tab to view the message log.

Figure 43.44 shows the Message Log panel. Click on the flashlight to view the messages from the latest run. We see the Run Control parameters (`Crw Online DUMMY`) and each table with the number of rows processed. Let's use our database query tool again to see if the rows have been deleted.

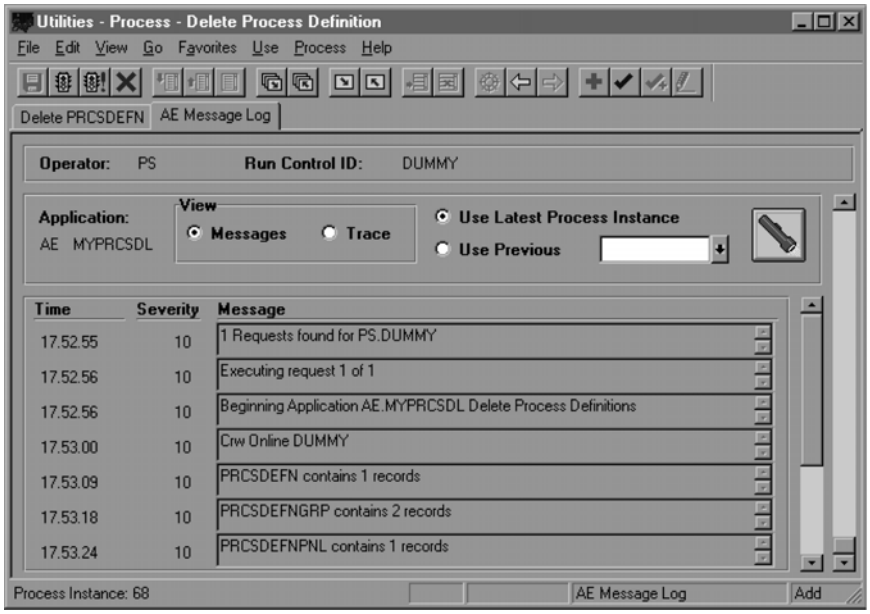


Figure 43.44 Reviewing the message log

43.2.1 Verifying our results

We verified that the DUMMY process definition rows were removed from the process definition tables using SQL*Talk (figure 43.45). Each `Select` returned zero rows. We can also examine the trace file for a more detailed look at the results.

You'll find the trace file in the `%TEMP%/ps/<databasename>` directory. The filename <process_instance>.aet will be used. In our particular case that translates to

```
C:\windows\temp\ps\hrdmo\68.aet
```

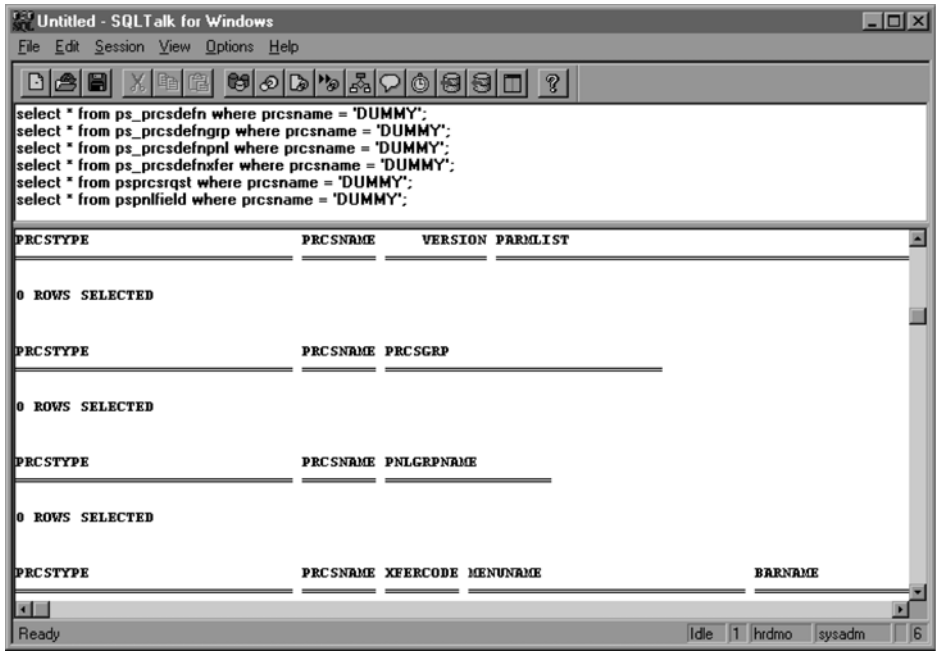


Figure 43.45 Verifying our test results using SQL*Talk

43.2.2 Examining the trace file

Let's take a look at the trace file contents:

Listing 43.1

The trace file

```
17.52.56 1999-09-04 PeopleTools 7.5 Application Engine
17.52.56 Tracing request PS.DUMMY
17.52.56 Starting application AE.MYPRCSDL Delete Process Definitions
/
INSERT INTO PS_USER_AET ( PROCESS_INSTANCE,COUNTER,RECNAME,FIELDNAME,
AE_DECIDE,PRCSTYPE,PRCSNAME,AE_SECTION  )
 VALUES (         68,0,' ',' ',' ',' ',' ',' '  )
/
COMMIT
/
17.52.56 .(AE.MYPRCSDL.MAIN.STEP1) (SELECT)
/
SELECT B.PRCSTYPE,B.PRCSNAME
  FROM PS_AE_RUN_CONTROL A,               PS_MY_RUN_CNTL_AE B
 WHERE A.OPRID = B.OPRID
   AND A.RUN_CNTL_ID = B.RUN_CNTL_ID
   AND A.PROCESS_INSTANCE = 68
```

```
/
17.52.57 .(AE.MYPRCSDL.MAIN.STEP2) (UPDATE)
17.53.01 .(AE.MYPRCSDL.MAIN.STEP3) (DO AE.MYPRCSDL.DYNSECTN)
/
SELECT 'PROCESS1', RECNAME
  FROM PSRECDEFN
 WHERE RECNAME = 'PRCSDEFN'
    OR RECNAME = 'PRCSDEFNGRP'
    OR RECNAME = 'PRCSDEFNPNL'
    OR RECNAME = 'PRCSDEFNXFER'
  UNION
 SELECT 'PROCESS2', RECNAME
  FROM PSRECDEFN
 WHERE RECNAME = 'PSPRCSRQST'
    OR RECNAME = 'PSPNLFIELD'
  ORDER BY 1, 2
/
17.53.03 ..(AE.MYPRCSDL.DYNSECTN.STEP1) (DO AE.MYPRCSDL.&SECTION)
17.53.03 ...(AE.MYPRCSDL.PROCESS1.STEP1) (SELECT)
/
SELECT COUNT(*)
  FROM PS_PRCSDEFN
 WHERE PRCSTYPE = 'Crw Online'
   AND PRCSNAME = 'DUMMY'
/
17.53.04 ...(AE.MYPRCSDL.PROCESS1.STEP2) (DO AE.MYPRCSDL.DELETE1)
/
SELECT 'X'
  FROM PSLOCK
 WHERE 1 > 0
/
17.53.07 ....(AE.MYPRCSDL.DELETE1.STEP1) (UPDATE)
/
DELETE
  FROM PS_PRCSDEFN
 WHERE PRCSTYPE = 'Crw Online'
   AND PRCSNAME = 'DUMMY'
/
17.53.08 ...(AE.MYPRCSDL.PROCESS1.STEP3) (DO AE.MYPRCSDL.MESSAGE)
17.53.09 ....(AE.MYPRCSDL.MESSAGE.STEP1) (UPDATE)
17.53.11 .(AE.MYPRCSDL.MAIN.STEP3) (DO FETCH)
17.53.11 ..(AE.MYPRCSDL.DYNSECTN.STEP1) (DO AE.MYPRCSDL.&SECTION)
17.53.12 ...(AE.MYPRCSDL.PROCESS1.STEP1) (SELECT)
/
SELECT COUNT(*)
  FROM PS_PRCSDEFNGRP
 WHERE PRCSTYPE = 'Crw Online'
   AND PRCSNAME = 'DUMMY'
/
17.53.14 ...(AE.MYPRCSDL.PROCESS1.STEP2) (DO AE.MYPRCSDL.DELETE1)
/
SELECT 'X'
  FROM PSLOCK
```

```
 WHERE 2 > 0
/
17.53.15 ....(AE.MYPRCSDL.DELETE1.STEP1) (UPDATE)
/
DELETE
  FROM PS_PRCSDEFNGRP
 WHERE PRCSTYPE = 'Crw Online'
   AND PRCSNAME = 'DUMMY'
/
17.53.17 ...(AE.MYPRCSDL.PROCESS1.STEP3) (DO AE.MYPRCSDL.MESSAGE)
17.53.17 ....(AE.MYPRCSDL.MESSAGE.STEP1) (UPDATE)
17.53.19 .(AE.MYPRCSDL.MAIN.STEP3) (DO FETCH)
17.53.19 ..(AE.MYPRCSDL.DYNSECTN.STEP1) (DO AE.MYPRCSDL.&SECTION)
17.53.19 ...(AE.MYPRCSDL.PROCESS1.STEP1) (SELECT)
/
SELECT COUNT(*)
  FROM PS_PRCSDEFNPNL
 WHERE PRCSTYPE = 'Crw Online'
   AND PRCSNAME = 'DUMMY'
/
17.53.21 ...(AE.MYPRCSDL.PROCESS1.STEP2) (DO AE.MYPRCSDL.DELETE1)
/
SELECT 'X'
  FROM PSLOCK
 WHERE 1 > 0
/
17.53.22 ....(AE.MYPRCSDL.DELETE1.STEP1) (UPDATE)
/
DELETE
  FROM PS_PRCSDEFNPNL
 WHERE PRCSTYPE = 'Crw Online'
   AND PRCSNAME = 'DUMMY'
/
17.53.23 ...(AE.MYPRCSDL.PROCESS1.STEP3) (DO AE.MYPRCSDL.MESSAGE)
17.53.23 ....(AE.MYPRCSDL.MESSAGE.STEP1) (UPDATE)
17.53.25 .(AE.MYPRCSDL.MAIN.STEP3) (DO FETCH)
17.53.25 ..(AE.MYPRCSDL.DYNSECTN.STEP1) (DO AE.MYPRCSDL.&SECTION)
17.53.26 ...(AE.MYPRCSDL.PROCESS1.STEP1) (SELECT)
/
SELECT COUNT(*)
  FROM PS_PRCSDEFNXFER
 WHERE PRCSTYPE = 'Crw Online'
   AND PRCSNAME = 'DUMMY'
/
17.53.27 ...(AE.MYPRCSDL.PROCESS1.STEP2) (DO AE.MYPRCSDL.DELETE1)
/
SELECT 'X'
  FROM PSLOCK
 WHERE 1 > 0
/
17.53.28 ....(AE.MYPRCSDL.DELETE1.STEP1) (UPDATE)
/
```

```
DELETE
  FROM PS_PRCSDEFNXFER
 WHERE PRCSTYPE = 'Crw Online'
   AND PRCSNAME = 'DUMMY'
/
17.53.29 ...(AE.MYPRCSDL.PROCESS1.STEP3) (DO AE.MYPRCSDL.MESSAGE)
17.53.30 ....(AE.MYPRCSDL.MESSAGE.STEP1) (UPDATE)
17.53.31 .(AE.MYPRCSDL.MAIN.STEP3) (DO FETCH)
17.53.31 ..(AE.MYPRCSDL.DYNSECTN.STEP1) (DO AE.MYPRCSDL.&SECTION)
17.53.32 ...(AE.MYPRCSDL.PROCESS2.STEP1) (SELECT)
/
SELECT COUNT(*)
  FROM PSPNLFIELD
 WHERE PRCSTYPE = 'Crw Online'
   AND PRCSNAME = 'DUMMY'
/
17.53.42 ...(AE.MYPRCSDL.PROCESS2.STEP2) (DO AE.MYPRCSDL.DELETE2)
/
SELECT 'X'
  FROM PSLOCK
 WHERE 0 > 0
/
17.53.43 ...(AE.MYPRCSDL.PROCESS2.STEP3) (DO AE.MYPRCSDL.MESSAGE)
17.53.44 ....(AE.MYPRCSDL.MESSAGE.STEP1) (UPDATE)
17.53.45 .(AE.MYPRCSDL.MAIN.STEP3) (DO FETCH)
17.53.45 ..(AE.MYPRCSDL.DYNSECTN.STEP1) (DO AE.MYPRCSDL.&SECTION)
17.53.46 ...(AE.MYPRCSDL.PROCESS2.STEP1) (SELECT)
/
SELECT COUNT(*)
  FROM PSPRCSRQST
 WHERE PRCSTYPE = 'Crw Online'
   AND PRCSNAME = 'DUMMY'
/
17.53.47 ...(AE.MYPRCSDL.PROCESS2.STEP2) (DO AE.MYPRCSDL.DELETE2)
/
SELECT 'X'
  FROM PSLOCK
 WHERE 0 > 0
/
17.53.48 ...(AE.MYPRCSDL.PROCESS2.STEP3) (DO AE.MYPRCSDL.MESSAGE)
17.53.48 ....(AE.MYPRCSDL.MESSAGE.STEP1) (UPDATE)
17.53.49 .(AE.MYPRCSDL.MAIN.STEP3) (DO FETCH)
/
DELETE
  FROM PS_USER_AET
 WHERE PROCESS_INSTANCE = 0000000068
/
17.53.50 Application AE.MYPRCSDL ended normally
/
COMMIT
/
17.53.50 Application Engine ended normally
```

In this entire trace file for the run, you can see each section and step as it was executed as well as the compiled SQL statements. Take special note of the dynamic sections PROCESS1 and PROCESS2. Also, notice the resolved bind variables used in the run.

Our Application Engine development is complete. There is always a great feeling of accomplishment that accompanies the successful completion of an application. I would suggest taking a nice long break before moving on to chapter 44 (Additional topics). You deserve some relaxation after a job well done!

KEY POINTS

1. You should always determine the program structure before creating your program. Building the lower (subordinate) sections first and working backward will alleviate any step dependencies you may encounter.
2. Validate your program results thoroughly using your database query tool and the trace file output. Pay close attention to the resolved bind variables.
3. You may not have realized it, but you've taken some huge strides in learning one of PeopleSofts' up-and-coming tools, Expand your knowledge and experience by creating custom Application Engine programs to perform a variety of different tasks.

CHAPTER 44

Additional topics

So far, we've covered the basics of Application Engine, and we can surely begin developing batch processes. But what if your program does not yield the desired results? Or, worse yet, what if the process aborts unexpectedly. In this chapter, we'll discuss the use of trace files and learn how to restart an aborted process. In addition, we'll provide tips on tackling large, often cumbersome Application Engine programs. An SQR utility to help analyze A/E programs will be covered as well.

44.1 USING TRACE FILES

In chapter 35, we discussed the options available on the Application Definition panel. One of the options controls the creation of a trace file. The option indicates levels of detail to be included in the trace file. NO trace file is generated when the Trace option is set to `Off`. The trace filename is set to <process_instance>.AET and is placed in the current working directory. The trace file is simply an ASCII text file that displays each step as it is executed. Generated SQL may be written to the trace file.

Figure 44.1 shows the Application Definition panel. The Trace file option is turned `OFF`. The other trace options available are `SQL` and `Steps Only`. `SQL` displays all the steps executed along with the associated SQL statements. The `Steps Only` option only displays the steps executed. (No SQL statements are written to the trace file.)

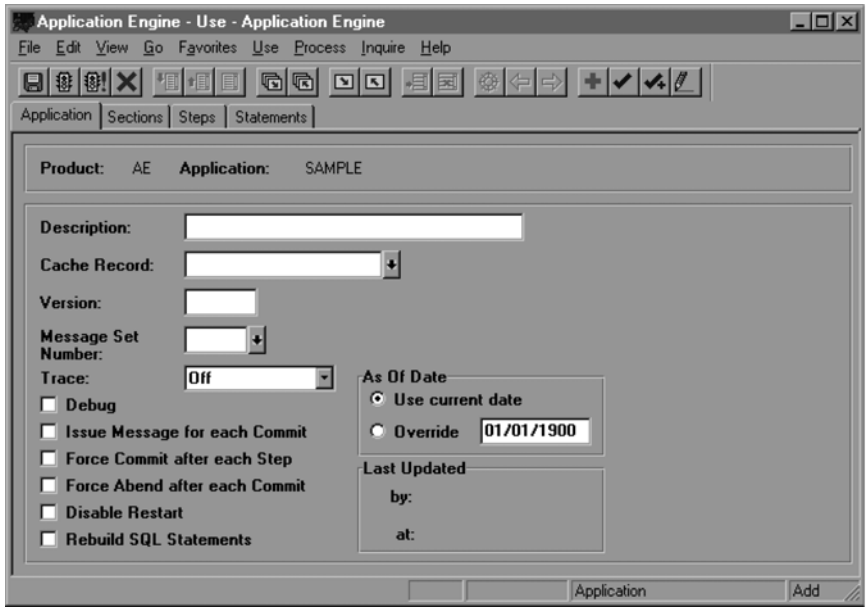


Figure 44.1 Trace options on Application Definition panel

44.1.1 Sample trace file

This sample trace file was generated during an execution of our exercise 5 application (called AE.USER005). The trace option was set to SQL, which displays all the steps and SQL statements processed. You can see some of the SQL statements are controlled by the PTPEMAIN process:

```
08.28.54 1999-01-13 PeopleTools 7.5 Application Engine
08.28.54 Tracing request PS.#USER005
08.28.54 Starting application AE.USER005 User Application 005
```

```
/
INSERT INTO PS_USER_AET ( PROCESS_INSTANCE,COUNTER,RECNAME,FIELDNAME,
AE_DECIDE  )
 VALUES (         10,0,' ',' ',' '  )
/
```

The `Insert` statement immediately preceding is an example of a PTPEMAIN-controlled statement. It initializes the cache record we specified on the Application Definition panel. Fields are initialized depending on their datatype. PROCESS_INSTANCE is always set to the process instance of our program. The field, COUNTER, is initialized to zero since it is numeric. The remaining fields in the cache record are initialized to blank since they are character datatypes.

```
UPDATE PS_USER_AET
   SET FIELDNAME = 'PAY_END_DT'
 WHERE PROCESS_INSTANCE = 0000000010
/
COMMIT
/
```

The statement above is also generated by PTPEMAIN. Remember, we initialized the cache field FIELDNAME to a value of `'PAY_END_DT'` on the Process Request panel. The value is then loaded into our cache record. This is how process request parameters are passed to the program.

The step in the code below selects `RECNAME` values and for each row returned (or fetched) executes a section called COUNT. The resolved bind variable for FIELDNAME is displayed in the trace file, and the value is set to `'PAY_END_DT'`.

```
08.28.55 .(AE.USER005.MAIN.STEP1) (DO AE.USER005.COUNT)
/
SELECT A.RECNAME
  FROM PSRECFIELD    A,
       PSRECDEFN     B
 WHERE A.RECNAME   = B.RECNAME
   AND A.FIELDNAME = 'PAY_END_DT'
   AND B.RECTYPE   = 0
  ORDER BY A.RECNAME
/
```

The RECNAME passed to COUNT.STEP1 (in this instance) is BEN_PLAN_DATA:

```
08.28.58 ..(AE.USER005.COUNT.STEP1) (SELECT)
/
SELECT COUNT(*)
  FROM PS_BEN_PLAN_DATA
/
08.29.00 ..(AE.USER005.COUNT.STEP2) (DO AE.USER005.MSG)
/
SELECT 'X'
```

```
  FROM PSLOCK
 WHERE 0 > 0
/
```

Notice all `&BIND()` variables have been resolved in the SQL. STEP1 selects the row count for the BEN_PLAN_DATA table. COUNT.STEP2, as you may recall, performs a `DO When` statement. If the number of rows is greater than zero, the section MSG is performed. The `WHERE` clause against the PSLOCK table may look odd at first. In our statement definition, the `WHERE` clause was defined as `WHERE &BIND(COUNTER) > 0`. Since the cache field COUNTER contains a value of zero, the statement was compiled as `WHERE 0 > 0`. This returns NO rows, and the MSG section is not executed. Control once again is passed to MAIN.STEP1, and another row is fetched:

```
08.29.02 .(AE.USER005.MAIN.STEP1) (DO FETCH)
08.29.03 ..(AE.USER005.COUNT.STEP1) (SELECT)
/
SELECT COUNT(*)
  FROM PS_BOND_LOG
/
08.29.05 ..(AE.USER005.COUNT.STEP2) (DO AE.USER005.MSG)
/
SELECT 'X'
  FROM PSLOCK
 WHERE 471 > 0
/
08.29.07 ...(AE.USER005.MSG.STEP1) (UPDATE)
```

This time a RECNAME value of `BOND_LOG` is passed to the COUNT section. There were 471 rows in the table. The `DO When` criteria once resolved reads as `WHERE 471 > 0`. This returns a row from PSLOCK, which designates a `TRUE` condition. The section MSG is then executed:

```
08.29.09 .(AE.USER005.MAIN.STEP1) (DO FETCH)
08.29.10 ..(AE.USER005.COUNT.STEP1) (SELECT)
/
SELECT COUNT(*)
  FROM PS_DED_CALC
/
08.29.13 ..(AE.USER005.COUNT.STEP2) (DO AE.USER005.MSG)
/
SELECT 'X'
  FROM PSLOCK
 WHERE 90 > 0
/
08.29.15 ...(AE.USER005.MSG.STEP1) (UPDATE)
/
```

This process continues until no rows remain. We'll skip the rest and proceed to the end of the trace file:

```
/
08.38.24 .(AE.USER005.MAIN.STEP1) (DO FETCH)
/
DELETE
  FROM PS_USER_AET
 WHERE PROCESS_INSTANCE = 0000000010
/
08.38.24 Application AE.USER005 ended normally
/
COMMIT
/
08.38.25 Application Engine ended normally
```

The last DO FETCH returned no rows, so the COUNT section was no longer executed. Our defined program has completed, and PTPEMAIN does some final cleanup by deleting the cache record row we've been using.

44.2 RESTARTING AN A/E PROCESS

During the execution of your A/E program, an unexpected error may cause it to abort. There are numerous reasons why this may occur: for example, an SQL error, system resource problem, or syntax errors in your A/E statement. A critical process could be near completion when the error occurs, and starting the process over from the beginning may not be a feasible solution. Application Engine maintains an entry in the AE_RUN_CONTROL table, which holds restart information in the event of an abend. This restart information is refreshed at every commit point. When restarting, the process takes over from the last commit point and continues. You cannot submit an aborted process from the beginning using the same OPRID and RUN_CNTL_ID. The AE_RUN_CONTROL holds it in a suspended status so it may be restarted properly.

Only applications run on the server through Process Scheduler can be restarted using Process Monitor by highlighting the failed process and clicking on Action → Restart. All other applications must execute PTPEMAIN.exe manually on the client or server when restarting. PTPEMAIN.exe should reside in the CBLBIN subdirectory attached to the PSVER directory (PSVER standing for the PeopleSoft version assigned as the high-level directory name). Simply type PTPEMAIN on the command line. You will then be prompted for the database type, database name, username (OPRID), password, Run Control ID, and process instance. The process will then resume where it left off.

There are times when you may wish to start the process from the beginning. If so, you'll have to delete the AE_RUN_CONTROL row for the OPRID and RUN_CNTL_ID you're using. You may also have to delete the cache record for the process instance. Only then can you restart the process from the beginning. You may also disable the restart capability on the Application Definition panel. Please make certain that no data corruption can occur as a result of not restarting properly.

44.3 ANALYZING A/E PROGRAMS

Analyzing large Application Engine programs can be tedious. One section can call a multitude of other sections. Since all components are stored within the database, you'll have to toggle back and forth between sections using the online panels. You can, however, take several steps that will make your analysis a little easier:

- Read any documentation on the process beforehand.
- Print a listing of the cache record used by the A/E program. This identifies all fields used to pass values from one step to another.
- Start with the section MAIN. Identify all steps within the MAIN section and treat each step as a separate process. Breaking the process down into smaller logical sections will help put things in perspective.
- Keep track of everything that's happening, not just the relationship between sections but also the tables being affected. Make a list of permanent tables and temporary tables. The temp tables can pass large amounts of data to subsequent sections or steps.
- After running the process, look at the trace file produced. One method of trace execution I've seen used allows you to sort the lines in the trace file using an ASCII editor (or import the lines into Excel or Access and then sort them). To do so, you remove any lines that don't have the time-stamp on it. You are then left with the time and name of all the steps performed in execution order. Be wary of processes that run beyond midnight. You'll have to manipulate the file in your editor to correct the sequencing. The resulting data will look like this:

```
08.28.54 1999-01-13 PeopleTools 7.5 Application Engine
08.28.54 Starting application AE.USER005 User Application 005
08.28.54 Tracing request PS.#USER005
08.28.55 .(AE.USER005.MAIN.STEP1) (DO AE.USER005.COUNT)
08.28.58 ..(AE.USER005.COUNT.STEP1) (SELECT)
08.29.00 ..(AE.USER005.COUNT.STEP2) (DO AE.USER005.MSG)
08.29.02 .(AE.USER005.MAIN.STEP1) (DO FETCH)
08.29.03 ..(AE.USER005.COUNT.STEP1) (SELECT)
08.29.05 ..(AE.USER005.COUNT.STEP2) (DO AE.USER005.MSG)
08.29.07 ...(AE.USER005.MSG.STEP1) (UPDATE)
08.29.09 .(AE.USER005.MAIN.STEP1) (DO FETCH)
08.29.10 ..(AE.USER005.COUNT.STEP1) (SELECT)
08.29.13 ..(AE.USER005.COUNT.STEP2) (DO AE.USER005.MSG)
08.29.15 ...(AE.USER005.MSG.STEP1) (UPDATE)
```

We'll skip the middle part of the trace and go to the end.

```
08.38.19 .(AE.USER005.MAIN.STEP1) (DO FETCH)
08.38.20 ..(AE.USER005.COUNT.STEP1) (SELECT)
08.38.22 ..(AE.USER005.COUNT.STEP2) (DO AE.USER005.MSG)
08.38.24 .(AE.USER005.MAIN.STEP1) (DO FETCH)
08.38.24 Application AE.USER005 ended normally
08.38.25 Application Engine ended normally
```

This method gives you a good idea of the execution flow of the program but it's not one hundred percent accurate. The first three lines were all processed at 08.28.54, but they are not in execution order. The third line, "Tracing request PS.#USER005," should come before the "Starting application…." line. Any lines with the same time-stamp will then sort alphabetically. When analyzing a specific portion of the program, this may not be a factor.

- A simple SQR program can be written to extract the time-stamped lines from the trace file. This would be more effective than manipulating the file in Excel or Access. Basic SQR skills would be required.
- When using a trace file, remember a trace file only displays the steps based on certain conditions met at the time. Running the process two different times could follow two different execution paths. The trace files are simply "after the fact." PeopleSoft does not provide a mechanism to produce an indented "tree-formatted" flowchart of an entire A/E process. The next portion of this chapter deals with a custom SQR utility I developed to analyze an Application Engine program. The cache record is listed; each step is listed with all of its defined attributes; and, finally, an indented process flowchart is produced which illustrates the full execution flow of the process. I have named this utility (appropriately) "Application Engine Analyzer."

44.4 APPLICATION ENGINE ANALYZER

I developed this SQR utility to extract all the required information to analyze an A/E program. Once the program is identified through user-entered prompts, several functions are performed. The cache record is listed; the steps are printed with their attributes; and an indented process flowchart is produced.

When running the SQR utility, make sure the communication box is visible by using the –CB option (figure 44.2). This displays additional information before each prompt.

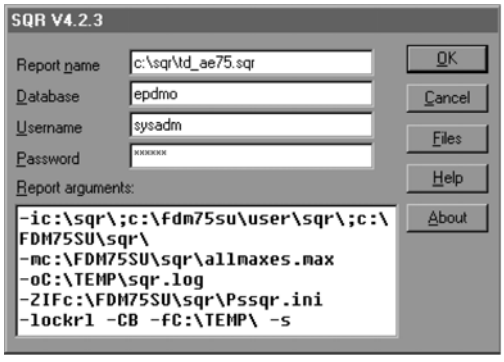


Figure 44.2
Running the SQR utility

The basic prompts are product, application ID and database platform. Figure 44.3 shows an example of the prompts issued when running the Application Analyzer utility. The product entered is AR, which is the product ID for Accounts Receivable. The application ID is PREDICT, the Payment Predictor process found in Accounts Receivable. This example was run under SQLBASE, so I entered the SQLBASE code of "1" for the database platform prompt. When a step has a SQLBASE-specific version, the utility substitutes that in place of the generic one. This is how Application Engine processes steps as well. An additional prompt creates an A/E statement file, which produces a temporary file that can be imported into Word, formatted, and printed. It includes extracted A/E statements also in execution order. This is extremely valuable information put in an organized fashion. Only use this feature when absolutely necessary—the output can be very large, slowing the process down.

Figure 44.3 SQR utility prompts for user information

The user is also prompted when a dynamic DO section (&SECTION) is encountered. By entering the section #DETAIL, the dynamic DO is resolved and included in the process. #DETAIL is a payment predictor matching algorithm that may be called dynamically when running the payment predictor process. You may substitute any valid algorithm for the &SECTION substitution variable.

Let's now examine some sample output which has been condensed for the book:

```
Report ID:  TD_AE75            APPLICATION ENGINE ANALYZER
    Phase:  1
  Program:  AR/PREDICT          Cache Record: PP_AET
================================================================================
Fieldname               Key     Type     Len     Dec     LongName
================================================================================


PROCESS_INSTANCE        Y       Nbr       10             Process Instance
OPRID                   N       Char       8             Operator Id
RUN_CNTL_ID             N       Char      30             Run Control ID
SETID                   N       Char       5             SetID
PP_METHOD               N       Char      15             Payment Predictor Method
EFFDT                   N       Date      10             Effective Date
PP_SEQ_NUM              N       Nbr        3             Sequence
PP_USAGE                N       Char       1             Usage
PP_SORT_SEQ_NUM         N       Nbr        3             Sorting Sequence number
```

The AR.PREDICT Application Engine program (known as Payment Predictor) uses a cache record called PP_AET. The first portion of the utility prints each field in PP_AET along with the field attributes.

The second portion lists each step in the A/E program with the MAIN section listed first. Additional attributes, such as DO section, Step Information, DO types, DB Platform, and so forth, are listed:

```
Report ID:  TD_AE75            APPLICATION ENGINE ANALYZER
    Phase:  2
  Program:  AR/PREDICT
================================================================================
Section  Step      Do         Activity               Type Update Select DO_When
================================================================================


MAIN     INIT      INIT       (DO INIT)              D    N      N      N
MAIN     PREP      PREP       (DO PREP)              D    N      N      Y
MAIN     SBLD      SBLD       (DO SBLD)              D    N      N      Y
MAIN     UPDM      UPDM       (DO UPDM)              D    N      N      Y
MAIN     DOC_SEQ   DOCSEQ     (DO DOCSEQ)            D    N      N      N
MAIN     REALGAIN  REALGAIN   (DO REALGAIN)          D    N      N      N
MAIN     PGEN      PGEN       (DO PGEN)              D    N      N      Y
MAIN     PUPD      PUPD       (DO PUPD)              D    N      N      Y
MAIN     TERMINAT  TERMINAT   (DO TERMINAT)          D    N      N      N
```

Let's look at steps that execute a COBOL process and a dynamic DO section. Notice the `&SECTION` has been resolved with the user-entered section #DETAIL.

```
DOC-CBL  FTPDOCAE              (COBOL: FTPDOCAE)       C    N      N      N

STEP     ALGR      #DETAIL     (DO &SECTION:#DETAIL)   D    N      N      Y
```

The process flow portion of the SQR utility begins with the section MAIN and flowcharts all steps in execution order. If a section has already been analyzed, the literal <Repeated Section> appears after it. There is no need to drill-down a second time:

```
Report ID:  TD_AE75            APPLICATION ENGINE ANALYZER
    Phase:  3
  Program:  AR/PREDICT
================================================================================
Process Flowchart
================================================================================


 ....(MAIN.INIT)  (DO INIT)
 .........(INIT.STARTMSG)  (UPDATE)
 .........(INIT.ROUNDSET)  (SELECT)
 .........(INIT.ROUNDIN3)  (DO ROUNDIN3)
 ..............(ROUNDIN3.DECIMAL3)  (SELECT)
 .........(INIT.REQUESTS)  (UPDATE)
 .........(INIT.CNT)  (SELECT)
 .........(INIT.MSG)  (UPDATE)
 .........(INIT.NONE)  (DO MSG_NONE)
 ..............(MSG_NONE.MESSAGE)  (UPDATE)
 ....(MAIN.PREP)  (DO PREP)
 .........(PREP.CLEARTMP)  (DO CLEARTMP)
 ..............(CLEARTMP.PAYMENT)  (UPDATE)
 ..............(CLEARTMP.CUST)  (UPDATE)
 ..............(CLEARTMP.ITEM)  (UPDATE)
 ..............(CLEARTMP.ITEM2)  (UPDATE)
 ..............(CLEARTMP.MATCH)  (UPDATE)
 .........(PREP.MESSAGE)  (UPDATE)
 .........(PREP.PAYMENTS)  (UPDATE)
 .........(PREP.DOC_SEQ)  (UPDATE)
 .........(PREP.SET_REF)  (UPDATE)
 .........(PREP.ID_ITEM)  (DO ID_ITEM)
 ..............(ID_ITEM.CUSTMP1)  (UPDATE)
 ..............(ID_ITEM.DUPE1)  (DO DUPECUST)
 ...................(DUPECUST.DEL_DUPE)  (UPDATE)
 ...................(DUPECUST.COPYTMP2)  (UPDATE)
 ...................(DUPECUST.CLEANUP)  (UPDATE)
 ..............(ID_ITEM.CUSTMP2)  (UPDATE)
 ..............(ID_ITEM.DUPE2)  (DO DUPECUST)     <Repeated Section>
```

Here is the portion of the process flowchart where the dynamic DO is encountered. The #DETAIL section entered by the user has been substituted and included in the listing:

```
............(STEP.ALGR)  (DO &SECTION:#DETAIL)
................(#DETAIL.ALGO_1)  (SELECT)
................(#DETAIL.ADJUST)  (DO ADJUST)
....................(ADJUST.INIT)  (UPDATE)
....................(ADJUST.ADJ_OVER)  (UPDATE)
....................(ADJUST.ADJ_UNDR)  (UPDATE)
....................(ADJUST.NAMEOVER)  (DO AUTO_ADJ)  <Repeated Section>
....................(ADJUST.NAMEUNDR)  (DO AUTO_ADJ)  <Repeated Section>
....................(ADJUST.COPYTMP2)  (DO COPYTMP2)  <Repeated Section>
................(#DETAIL.MATCHTMP)  (UPDATE)
................(#DETAIL.ALGR_B1)  (DO DUPES)
....................(DUPES.DUPES)  (DO ALGR_DUP)
..........................(ALGR_DUP.GET_ONE)  (SELECT)
..........................(ALGR_DUP.DEL_REST)  (UPDATE)
................(#DETAIL.ALGR_C1)  (DO PYSTATUS)
```

Let's look at how the COBOL section appears in the process flowchart. The developer can easily identify any COBOL or Mass Change sections:

```
....(MAIN.DOC_SEQ)  (DO DOCSEQ)
.........(DOCSEQ.DOC_SEQ)  (DO DOC_SEQ)
..............(DOC_SEQ.CHK_SEQ)  (DO DOC-CBL)
...................(DOC-CBL.FTPDOCAE)  (COBOL: FTPDOCAE)
..............(DOC_SEQ.SETID)  (SELECT)
..............(DOC_SEQ.GET_TYPE)  (SELECT)
..............(DOC_SEQ.UPD_BU)  (UPDATE)
..............(DOC_SEQ.GET_SEQ)  (DO DOC-CBL)     <Repeated Section>
....(MAIN.REALGAIN)  (DO REALGAIN)
```

44.4.1 Application Engine Analyzer source code—TD_AE75.SQR

The Application Engine Analyzer program processes sections and steps in the same manner as the PTPEMAIN process. The complete source code may be downloaded from the website http://www.sqrtools.com (under Utilities).

This process has been tested under Oracle, SQLBase, and DB2, but it may work with other databases as well. Additional updates may be posted to SQRTOOLS.COM, which may include compatibility with non-compliant databases.

Versions prior to Application Engine 7.5 are supported as well. Simply deactivate the substitution variable AE_75. This will bypass all references to the columns AE_DO_PRODUCT and AE_DO_APPLID. A major (and quite useful) enhancement in version 7.5 was the ability to call sections outside of the current application. The two aforementioned columns allow a called section to be qualified with the product and application ID, if necessary.

KEY POINTS

1. Use the Trace option to generate trace files for the Application Engine program. The trace file will show you the steps performed along with the SQL statements and resolved bind variables.
2. You can restart an Application Engine program so it picks up at the last commit point before it failed. This helps maintain system integrity when a process aborts.

CHAPTER 45

Application Engine—PeopleSoft 8

As we have proceeded through each Application Engine chapter, we've covered more concepts of Application Engine development. As an SQL processing tool, A/E can be used to create efficient batch processes. A/E's many useful features include decision capability and loop control, dynamic section calling, and messaging functionality. Other nice features in the current release are the trace file generation and the Commit/Restart logic (when a process terminates abnormally). All said, A/E is an extremely useful and well-conceived tool.

45.1 APPLICATION ENGINE "WISH LIST"

Although Application Engine is undeniably a tribute to creativity and resourcefulness, one can't help but think of enhancements that might still be made to the existing product. For instance, Application Engine exclusively acts on data that resides in the database itself. Imagine if Application Engine had the capability to read or write external files. This would make Application Engine an ideal choice for interface and conversion applications.

The Application Engine Definition panels are adequate for developing your programs, but a more intuitive graphical interface would be more suitable. The ability to view your program as a tree structure with each section and step in execution order would be a tremendous help.

There are times when updating a simple cache field value may seem cumbersome. Using the `&SELECT` function against the PSLOCK table (or any other single row table) is an ingenious solution, but is a bit convoluted. It also requires an additional call to the database where an alternative method may not need to do so. Application Engine could also use a mechanism to handle complex `IF-THEN-ELSE` expressions.

The ability to add Application Engine components to a project would be a welcome enhancement. Customizations could then be managed the same as other PeopleTool objects. Also, having Change Control in effect to lock your Application Engine programs would prevent other users from concurrently updating your program.

The types of enhancements I've mentioned here would elevate Application Engine to a much higher level, making it difficult to ignore the batch-processing capability that Application Engine provides. Let's now take a look at some of the great features implemented in release 8, some of which are nothing short of spectacular.

45.2 PEOPLESOFT RELEASE 8

Release 8 of PeopleSoft contains all of our "wish list" enhancements plus many additional features that can make Application Engine the tool of choice for many business processes. The single most important feature is Application Engines' complete integration with Application Designer. All Application Engine components are now objects. This means they can be placed into projects just as a record or panel definition would. You can also utilize the Change Control functionality to lock and unlock the Application Engine objects you're working on. Application Engine is now written entirely in C++. COBOL is no longer used to execute A/E programs.

When creating or modifying Application Engine objects, you will encounter a new graphical interface. It is much more intuitive than prior versions and behaves in a fashion similar to PeopleTool object interfaces. The Application Engine program is displayed in Definition or Program Flow view. The Program Flow view allows you to view your Application Engine program as a tree structure with each section and step displayed as a tree node. You can click on the "+" or "-" to expand or collapse a node. Any object type actions within a step are also displayed. Object types include SQL

`Selects` (DO types), SQL objects, other A/E sections, Message Log, and PeopleCode objects. Yes, that's correct—PeopleCode! Application Engine can now invoke PeopleCode and share many common business functions with online PeopleCode. A/E can be used to update fields in a state record (formerly referred to as a cache record). Any complex `IF-THEN-ELSE` expressions can be written in PeopleCode as well.

A new set of PeopleCode functions and classes have been added to support Application Engine. Some allow the reading and writing of external files. A new PeopleCode File class has been created that allows a variety of file handling operations to take place. You can even define a file layout with the new file layout definition in Application Designer and utilize it in your program.

Application Engine functions and macros have been replaced with Meta-SQL and a new set of macros. The Meta-SQL set has been expanded for greater functionality. For example, system (Meta) variables, which serve as text substitution variables, have been introduced. An example of a Meta-Variable would be `%ProcessInstance`, which contains the process instance of the run. `%RunControl` contains the Run Control ID used for the run. Prior versions of Application Engine required a database call against the A/E Run Control table to retrieve these values.

Temporary tables used in Application Engine programs have also increased functionality. Application Designer allows you to specify if a table is temporary. If so, you may designate the number of temporary table instances. For example, if you have a record called MY_TEMP, defined as a temporary table with three instances, the following physical SQL tables are created: PS_MY_TEMP, PS_MY_TEMP01, PS_MY_TEMP02, and PS_MY_TEMP03. During the execution of the Application Engine program, a specific instance of the temporary table can be utilized. This can greatly improve efficiency when running parallel processes.

Another interesting feature added is the Access checkbox on the section properties. If the section is designated as `Public`, all external Application Engine programs may call the section. If it is not `Public`, then the section is not available to any other programs. This is an excellent security measure that will prevent sensitive and potentially destructive SQL statements from being executed inadvertently.

The Application Engine debugger is also introduced in PeopleSoft release 8. While using the debugger, you can set break points, step through the code, view and edit state record fields, and even switch to the PeopleCode debugger when executing PeopleCode actions. This is a great feature that will make testing and debugging your Application Engine programs much easier than in the past.

Let's take a quick tour of some of the Application Engine features in PeopleSoft release 8.

45.2.1 Application Designer—Creating

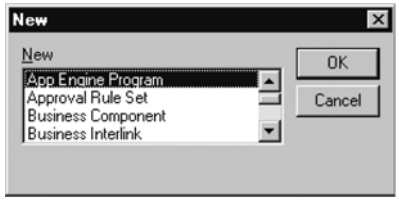


Figure 45.1 Creating a new Application Engine program

As mentioned previously, the Application Engine Designer Interface is accessed through Application Designer. You can create an Application Engine program by selecting File → New… and then selecting the Application Engine Program object in the drop down list (figure 45.1).

Once you select the new or existing Application Engine program, you can view or modify the program properties. Figure 45.2 shows the Program Properties panel. You can add a description and comments on the General folder tab.

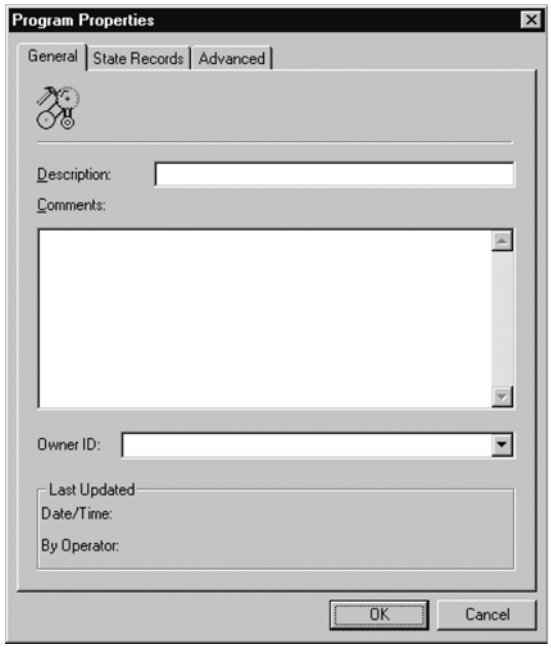


Figure 45.2
Program Properties—General

The State Record folder tab allows you to enter the State record(s) used by your Application Engine program. Multiple state records may be utilized by the program. You set the default state record by clicking on the Default State Record checkbox (figure 45.3). Note that the state record was formerly known as a cache record. The state record must still end with the suffix _AET as in prior versions.

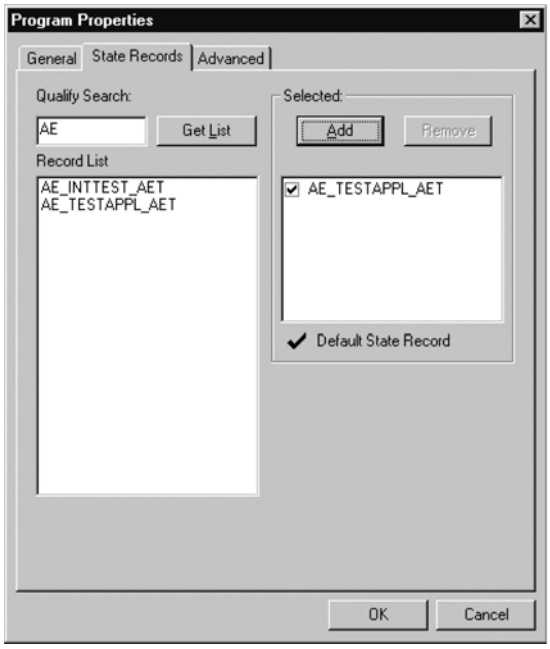


Figure 45.3
Program Properties—State Record

State records have much more functionality in release 8. They can now be used globally. The same state record can be used by both the calling and called program. Parameters can easily be passed from one program to another when sharing the same state record.

The Advanced folder tab (figure 45.4) lets you specify the default Message Set, Disable Restart, and designate Upgrade Only programs. In addition, your program can be defined as an Application Library. An Application Library is not an executable Application Engine program but a collection of sections that can be called by other Application Engine programs.

You make your actual program modifications using the Application Engine Definition interface. Two tabs allow you to view your program components: the definition view and the program flow view. The definition view (figure 45.5) allows you to create sections, steps, and actions, which are displayed as nodes. You can collapse and expand the nodes to drill down into each section. The sections in the definition view are not displayed in the order they are executed. You need to click on the Program Flow tab to view the execution order of the program.

Pay close attention to the Project Workspace window in figure 45.5. The Application Engine object has been inserted into the project. As you can see, Application Engine is fully integrated with PeopleTools in release 8.

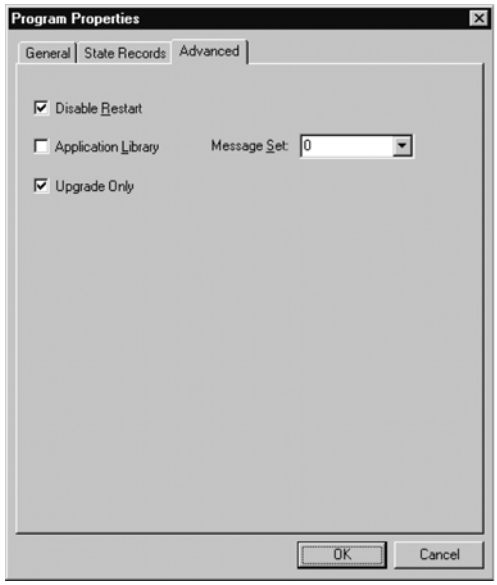


Figure 45.4
Program Properties—Advanced

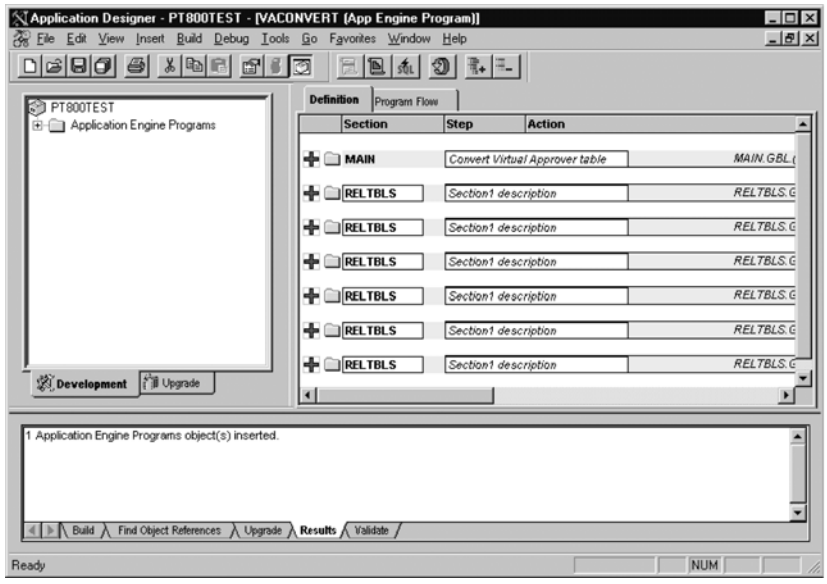


Figure 45.5 Application Engine Definition view

You can access the section properties by clicking on the section node and then clicking the right mouse button. Notice the Access checkbox in the section properties (figure 45.6). You can make the section `Public` by clicking the checkbox. Another

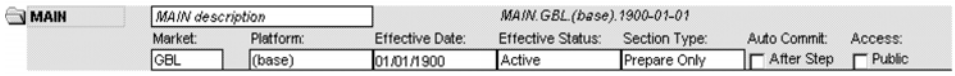


Figure 45.6 Viewing section properties

new feature is the Market designation. You can define your section as `Global` (`GBL`) or use a market code such as `USA` or `JPN` to make your section market specific.

The Application Engine Program Flow View (figure 45.7) displays the program as a tree structure with each node in its logical execution sequence. This feature should aid developers by providing a graphical representation of their Application Engine program.

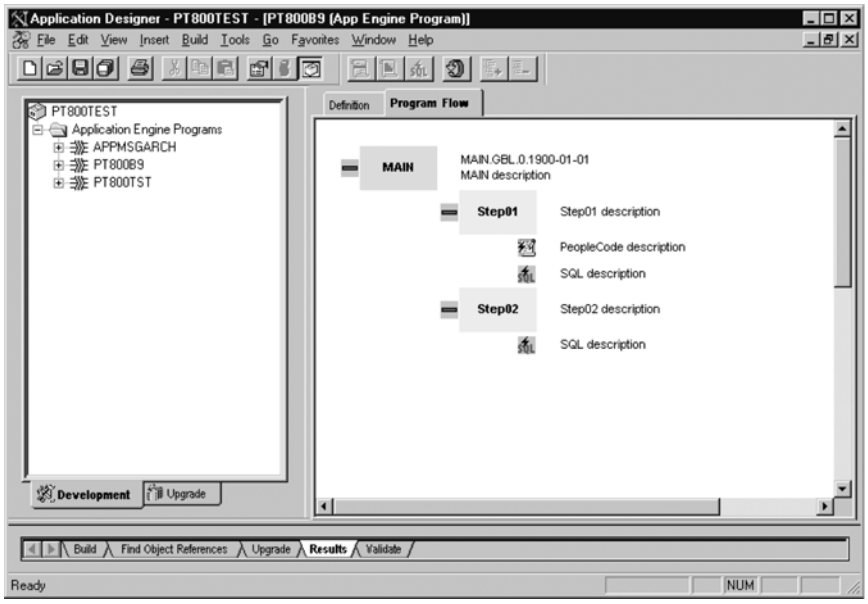


Figure 45.7 Application Engine Program Flow view

Notice the PeopleCode node under Step01 (figure 45.7). You can insert PeopleCode actions (in Definition view) within a step. You use the PeopleCode Editor to write the PeopleCode program.

You can invoke the PeopleCode Editor by double-clicking on the PeopleCode node (Figure 45.8). SQL actions can be inserted the same way within a step and modified using something called the SQL Editor. Each Action Type, viewed as a node, will have a particular set of action type properties.

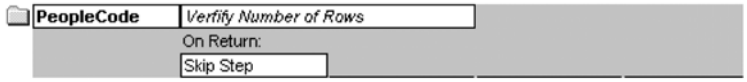


Figure 45.8 Accessing a PeopleCode program in Definition view

Figure 45.9 shows two action type nodes. The `Do Select` action type displays the description, `Reuse` statement, and `Do Select` type properties. The `Call Section` action type displays the description, section name, program ID, and dynamic section properties.

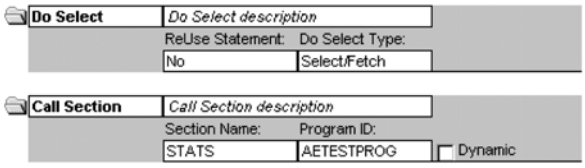


Figure 45.9 Action types and action type properties

45.2.2 Action types

Table 45.1 shows the possible action types along with the object type, the available properties, and the corresponding editor used to create the code behind the action. COBOL and Mass Change programs can no longer be called through the action properties. You can still call a COBOL program, but it must be invoked using PeopleCode and the `RemoteCall()` function. Mass Change programs are no longer supported—alternatives, such as the Application Engine Mass Change program, can be used instead. The message action is used in place of the `&MSG` function. The Message properties contain the same parameters as the `&MSG` function.

Table 45.1 Action types and associated properties

Action Type	Object Type	Properties	Editor
DO When	SQL Select	ReUse Statement	SQL Editor
DO While	SQL Select	ReUse Statement	SQL Editor
DO Until	SQL Select	ReUse Statement	SQL Editor
DO Select	SQL Select	ReUse Statement DO Select Type	SQL Editor
PeopleCode	PeopleCode	On Return	PeopleCode Editor
SQL	SQL Statement	ReUse Statement No Rows	SQL Editor

Table 45.1 Action types and associated properties (continued)

Action Type	Object Type	Properties	Editor
Call Section	A/E Section	Section Name Program ID Dynamic Section	N/A
Message	Message Log	Message Set Message Set Number Message Parameters	N/A

45.2.3 Meta-SQL

Application Engine now supports Meta-SQL such as `%DateIn` and `%DateOut`. Application Engine Meta-SQL constructs have been added.

Table 45.2

Function	Description
`%Bind`	Retrieves a value from the State record.
`%ExecuteEdits`	Supports data dictionary edits in batch mode. This includes any field defined with edit types of `Required`, `Yes/No`, `DateRange`, `Prompt Table`, or `Translate Table`. Meta-Variables `%Edit_Required`, `%Edit_YesNo`, `%Edit_DateRange`, `%Edit_PromptTable`, and `%Edit_TranslateTable` are used to specify the particular Edit(s) required. These Meta-Variables can be added together to produce combination edits on a field.
`%Select`	Selects fields and updates State Record values. If the SQL `Select` returns no rows, the state record fields are untouched.
`%SelectInit`	Selects fields and updates state record values. If no rows are returned, the state record fields are initialized.
`%SQL`	Allows an SQL object to be utilized in Application Engine SQL statements or PeopleCode regardless of differences in bind variable syntax between the two.
`%Table`	Returns the SQL table name for the record name specified. This eliminates the need to prefix certain tables with PS_ before accessing them. If the table is defined as a temporary table, the appropriate temporary table instance number is appended to the returned SQL table name (i.e., PS_MY_TEMPnn where nn is the instance number).
`%TruncateTable`	Depending on the database, either a `TRUNCATE TABLE` or `DELETE FROM` (without a `WHERE` clause) is generated.
`%UpdateStats`	Generates a platform-specific statement to update the system catalog tables for use in optimization procedures.

45.2.4 Application Engine macros

Some macros in release 8 look familiar. Some of the differences are in syntax only. The `&&RECORD` macro is no longer used.

Table 45.3 Application Engine macros

Macro	Description
`%ClearCursor`	recompiles re-used statements. Resets any `STATIC` `%Bind` variables
`%Execute`	execute database-specific commands such as PL/SQL Blocks
`%Next`	increments a sequence value
`%Previous`	decrements a sequence value
`%RoundCurrency`	rounds an amount to proper currency precision when using the Multi-Currency option

45.2.5 System Meta-Variables

Application Engine now provides useful Meta-Variables that eliminate unnecessary calls to the database to retrieve fields such as the Run Control ID and process instance. The SQL syntax is also simpler when using the Meta-Variables. `%Bind` is not needed to retrieve the values from the state record.

Table 45.4 Application Engine System Variables

Meta-Variable	Description
`%AeProgram`	current Application Engine program name (in quotes)
`%AeSection`	current Application Engine section name (in quotes)
`%AeStep`	current Application Engine step name (in quotes)
`%JobInstance`	Process Scheduler job instance number
`%ProcessInstance`	Process Instance
`%ReturnCode`	return code of last SQL statement
`%RunControl`	current Run Control ID (in quotes)
`%AsOfDate`	`As-Of-Date` of the current process (in quotes)
`%Comma`	character substitution— comma
`%LeftParen`	character substitution—left parenthesis
`%RightParen`	character substitution—right parenthesis
`%Space`	character substitution—space
`%SQLRows`	number of rows affected by SQL statement. `Select` statements return a value of 0 or 1 (to represent no rows or some rows, respectively).

45.2.6 Application Engine PeopleCode

The use of PeopleCode is one of the most powerful enhancements to Application Engine. You can update state records directly, perform complex `IF-THEN-ELSE` expressions, and process file input/output records. When you attach a PeopleCode

Action to a step, you may also specify the On-Return property, which is either `Abort`, `Break`, or `Skip Step`. The On-Return property is initiated when the PeopleCode program issues a `Non-Zero` or `True` return code. If no return code is assigned by the PeopleCode program, then zero is used as the default. `Abort` halts processing of the entire program. `Break` exits the entire section currently executing. `Skip Step` processes no additional actions attached to the current step—the next step is processed immediately.

Let's examine a few examples, drawing comparisons with PeopleSoft 7.5 when possible.

You can see that a database call is required in PeopleSoft 7.5 to update a single cache field value:

```
&SELECT(AE_SECTION)
SELECT 'PROCESS1'
  FROM PSLOCK
```

In the direct updating of the state record field using PeopleCode (PeopleSoft 8.0), no additional database call is required. Also, note the absence of a return code assignment which defaults to zero:

```
AE_SECTION = "PROCESS1";
```

Let's consider `IF-THEN-ELSE` logic now. Imagine that we need to execute a Mass SQL `Insert` only if the table into which we're inserting is empty. For this example, we assume the row count of our table into which has been determined and is contained in the field COUNTER (either in the cache or state record).

In PeopleSoft 7.5, a `DO When` statement type is used to execute an additional section, depending on the "SQL Select" results. If the COUNTER cache field is zero (meaning the table is empty), the `DO When` section specified (which performs the SQL Insert) is executed. If the COUNTER is not zero, the `DO When` section is not performed:

```
&SELECT(AE_DECIDE)
SELECT 'X'
  FROM PSLOCK
 WHERE &BIND(COUNTER) = 0
```

Look at the PeopleSoft 8 version using PeopleCode:

```
If USER_AET.COUNTER > 0
   Exit(1)
End-if;
```

The state record field COUNTER is interrogated directly. If the COUNTER field is greater than zero, the return code is set to `1` (or `TRUE`), and the On Return property of the PeopleCode action becomes effective. Let's assume the On Return property is set to Skip Step. Any additional actions for the current step are now bypassed

including the SQL action for our `Insert` statement. If the COUNTER is zero, the subsequent SQL action for the step is executed. (The Return Code defaults to zero in our PeopleCode action.)

PeopleSoft 8 enables file operations within Application Engine through PeopleCode. This is made possible by the new object classes now available. We'll demonstrate how PeopleCode actions can use these object classes to write records to a flat file. The output file will be created using a file layout definition. By changing the FileLayout property of our file object, we can switch file layouts whenever necessary.

We can now perform a simple demonstration for a typical outbound interface program. The sample PeopleCode program creates a flat file based on the contents of the table MY_TABLE. Let's assume this table was created during preceding steps of the Application. The columns are selected using a temporary SQL object created dynamically at run time. A Meta-SQL function (`%Selectall`) is used to build the `Select` statement. The SQL Object uses the `Fetch` method to retrieve each row one at a time. The file object (our output file) uses the file layout definition MY_LAYOUT:

```
Ln#  PeopleCode
---  -------------------------------------------------------
 1  Local Record &MY_REC;
 2  Local File   &MY_FILE;
 3  Local SQL    &MY_SQL;
 4
 5  &MY_FILE = GetFile("myoutput.txt", "W");
 6
 7  if &MY_FILE.IsOpen Then
 8     if &MY_FILE.SetFileLayout(FILELAYOUT.MY_LAYOUT) Then
 9        &MY_REC = CreateRecord(RECORD.MY_TABLE);
10        &MY_SQL = CreateSQL("%Selectall(:1)", &MY_REC);
11        While &MY_SQL.Fetch(&MY_REC)
12           &MY_FILE.WriteRecord(&MY_REC);
13        End-While;
14     End-If;
15  End-If;
16
17  &MY_FILE.Close();
18
```

Our program is displayed above with line numbers (for reference only). Let's take a closer look at each line in the PeopleCode program.

Lines 1 through 3 create temporary object variables: `&MY_REC` is a record object; `&MY_FILE` is a file object; and `&MY_SQL` is an SQL object. Each of these temporary variables has a set of properties unique to its own object type. We can now perform some simple manipulations to accomplish our task.

Line 5 uses the `GetFile` function to associate a file to our file object `&MY_FILE`. The `GetFile` function also opens "myoutput.txt" in Write Mode.

Line 7 tests to see if the file associated with `&MY_FILE` was opened successfully (evaluating the IsOpen property using dot notation).

Line 8 sets the FileLayout property of the `&MY_FILE` file object to our file layout definition (MY_LAYOUT).

Line 9 uses the `CreateRecord` function to pass the MY_TABLE attributes to the `&MY_REC` record object. Now MY_TABLE and `&MY_REC` have equivalent attributes.

Line 10 dynamically creates the SQL for the `&MY_SQL` object using the `CreateSQL` function. `%Selectall(:1)` is a Meta-SQL construct that creates the `Select` statement based on the record passed as a parameter. Since we passed the `&MY_REC` record object as the parameter, the record MY_TABLE is used. (Remember, `&MY_REC` now has the same attributes as MY_TABLE.)

Lines 11 through 13 perform a `Do While` loop. A `Fetch` method is performed using the SQL object we created (`&MY_SQL`). This selects each row one by one. The `WriteRecord` method for the `&MY_FILE` object is used with the `&MY_REC` object to write lines to the output file. The records are then written as directed by the file layout definition currently used.

The operations being performed may be considered complex, but the PeopleCode that is actually produced by the developer couldn't be much simpler. Also, note any changes to the original record or file layout definitions do not affect the PeopleCode program.

45.2.7 Application Engine debugger

Another exciting enhancement in release 8 is the Application Engine debugger. You must enable the debugger through Configuration Manager or as a command line option. You must also enable the PeopleCode debugger if you want to debug any PeopleCode actions in the Application Engine program. Debug mode is easy to use. Here's a glimpse of the Application Engine debugger Help menu:

```
PeopleTools 8.0 - Application Engine
Copyright (c) 1988-1999 PeopleSoft, Inc.
All Rights Reserved

Application Engine Debugger - enter command or type ? for help.

AEMYPRCSDL.MAIN.STEP1> ?
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
```

As you can see, the Application Engine debugger contains an extensive set of debug commands. The descriptions of the commands on the Help menu do not need much more elaboration, but I'd like to review a couple of commands.

The BREAK command allows you to set, and subsequently unset, breakpoints in your program. There is also an option to list the currently active breakpoints.

Let's consider how to set a breakpoint with the Set option. The user is prompted for the program, section, and step to which the breakpoint should be set:

```
AEMYPRCSDL.MAIN.STEP1> b
(S)et, (U)nset, or (L)ist? s
Program [AEMYPRCSDL]:
Section [MAIN]: DYNSECTN
Step    [STEP1]: STEP1

Breakpoint set at AEMYPRCSDL.DYNSECTN.STEP1
```

Now, let's look at the use of the Unset option. A list of active breakpoints is displayed along with a corresponding sequence number. The user must enter the sequence number of the breakpoint to remove it from the active breakpoint list. The List option displays the active breakpoint list without any additional options.

```
AEMYPRCSDL.MAIN.STEP1> b
(S)et, (U)nset, or (L)ist? u

Active Breakpoints:
(1) AEMYPRCSDL.MAIN.STEP2
(2) AEMYPRCSDL.DYNSECTN.STEP1

Remove which breakpoint? 1
```

The LOOK, MODIFY, and WATCH commands allow you to view and modify state record fields and designate watch fields. Once you set a watch field, the program stops when the value of the field changes.

```
Record Name [USER_AET]:
Field Name [*]:
USER_AET:
  PROCESS_INSTANCE   = 50
  COUNTER            = 1685
  RECNAME            = 'JOB'
  FIELDNAME          = ' '
  AE_DECIDE          = ' '
```

In our LOOK command, the record name selected was USER_AET (the default for this Application Engine program). All the fields in USER_AET are listed with their current values.

Consider now the results of our MODIFY command. We selected the COUNTER state record field and changed the value from 1685 to 0. The MODIFY command is a useful tool when testing conditions in your Application Engine program:

```
AEUSER003.MAIN.STEP2> m
Record Name [USER_AET]:
Field Name [none]: COUNTER

Current value:  USER_AET.COUNTER = 1685

Enter new value (do not use quotes around text strings):
0
```

The field RECNAME is selected as a watch field using the Set option of the WATCH command. The program will stop each time the value of this field changes. You can Unset and List watch fields in a similar manner as breakpoints.

```
Set or remove a watch field
AEUSER004.MAIN.STEP1> w
(S)et, (U)nset, or (L)ist? s
Record Name [USER_AET]:
Field Name [none]: RECNAME
```

If enabled, the PeopleCode debugger is invoked when a PeopleCode action is encountered. Many new features exist in the PeopleCode debugger such as Hover Inspect, where a pop-up displays the value of simple variables and fields simply by hovering over it with the mouse. The variable display window allows you to drill down on the properties of each object by expanding/collapsing the corresponding node. As you can see, the PeopleTools development team has been busy. The features I've mentioned in this chapter are just a small sampling of the next generation of PeopleTools!

Some readers may have come directly to this chapter to read about some of the great new Application Engine features in the PeopleSoft 8 release. If you have no familiarity with the PeopleSoft 7.5 version of Application Engine, I would suggest going through the tutorial in the preceeding chapters. The examples there will give you the opportunity to develop a good understanding of Application Engine concepts without being bombarded with terminology such as object classes, meta-this, meta-that, and such. The PeopleSoft 8 version of Application Engine builds and improves upon the concepts previously discussed. In this ever-changing world of technology, it's a good idea to take advantage of every learning opportunity you can.

A P P E N D I X A

Problem Tracking application

All objects used to build our Problem Tracking application are listed in this appendix. The readers can develop these objects as they read this book. The readers should not be limited to the objects in the appendix. They can further enhance the application and develop other objects by using the techniques described in this book.

Let us look at an ERD diagram of all the record definitions used to develop the Problem Tracking application. All columns which are in bold and underlined are part of the primary key. All columns in bold alone are alternate keys.

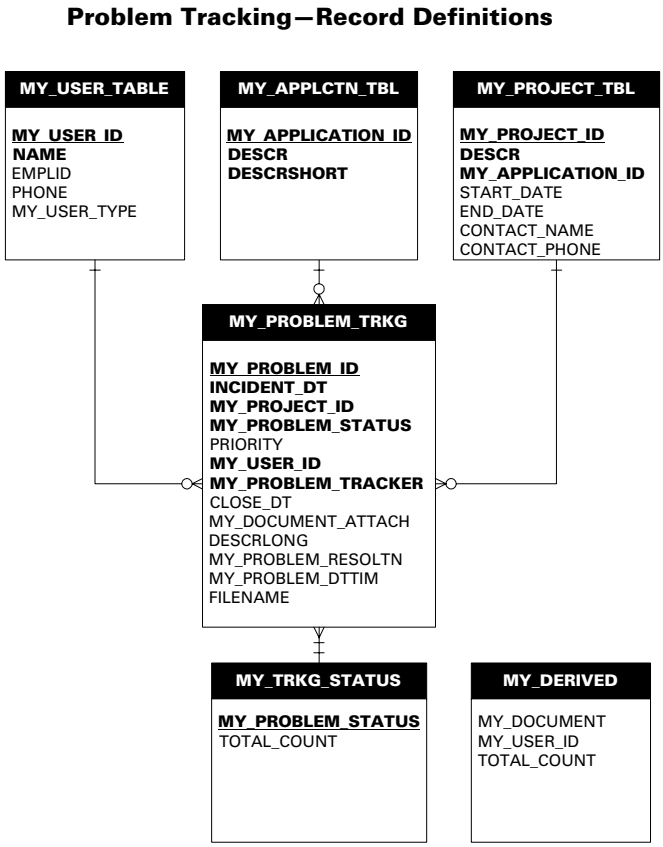


Figure A.1
Problem Tracking Application—record definitions

Let us list all the record definitions showing the different views. New fields start with a prefix of MY_. These fields have to be created in the system before the record definitions are built.

MY_USER_TABLE (Record)

Field Name	Type	Len	Format	H	Short Name	Long Name
MY_USER_ID	Char	6	Upper		User ID	User ID
NAME	Char	50	Name		Name	Name
EMPLID	Char	11	Upper		ID	EmplID
PHONE	Char	24	Custm		Phone	Telephone
MY_USER_TYPE	Char	1	Upper		User Type	User Type

Figure A.2 **MY_USER_TABLE Table—Field Display**

MY_USER_TABLE (Record)

Field Name	Type	Key	Dir	CurC	Srch	List	Sys	Audt	H	Default
MY_USER_ID	Char	Key	Asc		Yes	Yes	No			
NAME	Char	Alt	Asc		No	Yes	No			
EMPLID	Char	Alt	Asc		No	Yes	No			
PHONE	Char				No	No	No			
MY_USER_TYPE	Char				No	No	No			

Figure A.3 **MY_USER_TABLE Table—Use Display**

MY_USER_TABLE (Record)

Field Name	Type	Req	Edit	Prompt Table	Set Control Field	Rs Dt
MY_USER_ID	Char	Yes				No
NAME	Char	Yes				No
EMPLID	Char	No	Prompt	PERSONAL_DATA		No
PHONE	Char	No				No
MY_USER_TYPE	Char	No	Xlat			No

Figure A.4 **MY_USER_TABLE Table—Edits Display**

MY_USER_TABLE stores all the users reporting problems in our application.

MY_APPLCTN_TBL (Record)

Field Name	Type	Len	Format	H	Short Name	Long Name
MY_APPLICATION_ID	Char	3	Upper		Application	Application Identification
DESCR	Char	30	Mixed		Descr	Description
DESCRSHORT	Char	10	Mixed		Short Desc	Short Description

Figure A.5 MY_APPLCTN_TBL **Table—Field Display**

MY_APPLCTN_TBL (Record)

Field Name	Type	Key	Dir	CurC	Srch	List	Sys	Audt	H	Default
MY_APPLICATION_ID	Char	Key	Asc		Yes	Yes	No			
DESCR	Char	Alt	Asc		No	Yes	No			
DESCRSHORT	Char				No	No	No			

Figure A.6 MY_APPLCTN_TBL **Table—Use Display**

MY_APPLCTN_TBL (Record)

Field Name	Type	Req	Edit	Prompt Table	Set Control Field	Rs Dt
MY_APPLICATION_ID	Char	Yes				No
DESCR	Char	Yes				No
DESCRSHORT	Char	Yes				No

Figure A.7 MY_APPLCTN_TBL **Table—Edits Display**

MY_APPLCTN_TBL stores all applications that are tracked in our system.

MY_PROJECT_TBL (Record)

Field Name	Type	Len	Format	H	Short Name	Long Name
MY_PROJECT_ID	Char	6	Upper		Project ID	Project Identification
DESCR	Char	30	Mixed		Descr	Description
MY_APPLICATION_ID	Char	3	Upper		Application	Application Identification
START_DATE	Date	10			Start Date	Start Date
END_DATE	Date	10			End Date	End Date
CONTACT_NAME	Char	50	Mixed		Name	Contact Name
CONTACT_PHONE	Char	12	Phone		Phone	Contact Phone

Figure A.8 MY_PROJECT_TBL **Table—Field Display**

MY_PROJECT_TBL (Record)

Field Name	Type	Key	Dir	CurC	Srch	List	Sys	Audt	H	Default
MY_PROJECT_ID	Char	Key	Asc		Yes	Yes	No			
DESCR	Char	Alt	Asc		No	Yes	No			
MY_APPLICATION_ID	Char	Alt	Asc		No	Yes	No			
START_DATE	Date				No	No	No			
END_DATE	Date				No	No	No			
CONTACT_NAME	Char				No	No	No			
CONTACT_PHONE	Char				No	No	No			

Figure A.9 MY_PROJECT_TBL **Table—Use Display**

MY_PROJECT_TBL (Record)

Field Name	Type	Req	Edit	Prompt Table	Set Control Field	Rs Dt
MY_PROJECT_ID	Char	Yes				No
DESCR	Char	Yes				No
MY_APPLICATION_ID	Char	No	Prompt	MY_APPLCTN_TBL		No
START_DATE	Date	No				No
END_DATE	Date	No				No
CONTACT_NAME	Char	No				No
CONTACT_PHONE	Char	No				No

Figure A.10 MY_PROJECT_TBL **Table—Edits Display**

MY_PROJECT_TBL stores all projects that are tracked in our application.

MY_PROBLEM_TRKG (Record)

Field Name	Type	Len	Format	H	Short Name	Long Name
MY_PROBLEM_ID	Char	6	Num		Problem ID	Problem Identification
INCIDENT_DT	Date	10			Incdnt Dt	Incident Date
MY_PROJECT_ID	Char	6	Upper		Project ID	Project Identification
MY_PROBLEM_STATUS	Char	1	Upper		Problem Sta	Problem Status
PRIORITY	Nbr	3			Priority	Priority
MY_USER_ID	Char	6	Upper		User ID	User ID
MY_PROBLEM_TRACKEI	Char	6	Upper		Problem Tra	Problem Tracker
CLOSE_DT	Date	10			Close Date	Date Closed
MY_DOCUMENT_ATTACI	Char	1	Upper		Document?	Document Attached?
DESCRLONG	Long	0			Descr	Description
MY_PROBLEM_RESOLTI	Long	0			Prob.Resolu	Problem Resolution
MY_PROBLEM_DTTIM	DtTm	26	Scnds		Date/Time I	Date/Time Reported
FILENAME	Char	80	Mixed		File Name	File Name

Figure A.11 MY_PROBLEM_TRKG Table—Field Display

MY_PROBLEM_TRKG (Record)

Field Name	Type	Key	Dir	CurC	Srch	List	Sys	Audt	H	Default
MY_PROBLEM_ID	Char	Key	Asc		Yes	Yes	No			
INCIDENT_DT	Date	Alt	Asc		No	Yes	No			
MY_PROJECT_ID	Char	Alt	Asc		No	Yes	No			
MY_PROBLEM_STATUS	Char	Alt	Asc		No	Yes	No			'1'
PRIORITY	Nbr				No	No	No			
MY_USER_ID	Char	Alt	Asc		No	Yes	No			
MY_PROBLEM_TRACKEI	Char	Alt	Asc		No	Yes	No			
CLOSE_DT	Date				No	No	No			
MY_DOCUMENT_ATTACI	Char				No	No	No			
DESCRLONG	Long				No	No	No			
MY_PROBLEM_RESOLTI	Long				No	No	No			
MY_PROBLEM_DTTIM	DtTm				No	No	No			
FILENAME	Char				No	No	No			

Figure A.12 MY_PROBLEM_TRKG Table—Use Display

MY_PROBLEM_TRKG (Record)

Field Name	Type	Req	Edit	Prompt Table	Set Control Field	Rs Dt
MY_PROBLEM_ID	Char	Yes				No
INCIDENT_DT	Date	Yes				Yes
MY_PROJECT_ID	Char	No	Prompt	MY_PROJECT_TBL		No
MY_PROBLEM_STATUS	Char	No	Xlat			No
PRIORITY	Nbr	No				No
MY_USER_ID	Char	No	Prompt	MY_USER_TABLE		No
MY_PROBLEM_TRACKEI	Char	No	Prompt	MY_USER_TABLE		No
CLOSE_DT	Date	No				No
MY_DOCUMENT_ATTACI	Char	No	Y/N			No
DESCRLONG	Long	No				No
MY_PROBLEM_RESOLTI	Long	No				No
MY_PROBLEM_DTTIM	DtTm	No				No
FILENAME	Char	No				No

Figure A.13 MY_PROBLEM_TRKG Table—Edits Display

MY_PROBLEM_TRKG stores all incidents tracked in our application.

MY_TRKG_STATUS (Record)

Field Name	Type	Len	Format	H	Short Name	Long Name
MY_PROBLEM_STATUS	Char	1	Upper		Problem Sta	Problem Status
TOTAL_COUNT	Nbr	7			Total Cnt	Total Count

Figure A.14 MY_TRKG_STATUS View—Field Display

MY_TRKG_STATUS (Record)

Field Name	Type	Key	Dir	CurC	Srch	List	Sys	Audt	H	Default
MY_PROBLEM_STATUS	Char	Key	Asc		No	No	No			
TOTAL_COUNT	Nbr				No	No	No			

Figure A.15 MY_TRKG_STATUS View—Use Display

MY_TRKG_STATUS (Record)

Field Name	Type	Req	Edit	Prompt Table	Set Control Field	Rs Dt
MY_PROBLEM_STATUS	Char	No	Xlat			No
TOTAL_COUNT	Nbr	No				No

Figure A.16 MY_TRKG_STATUS View—Edits Display

MY_TRKG_STATUS is an SQL view that represents data from MY_PROBLEM_TRKG table.

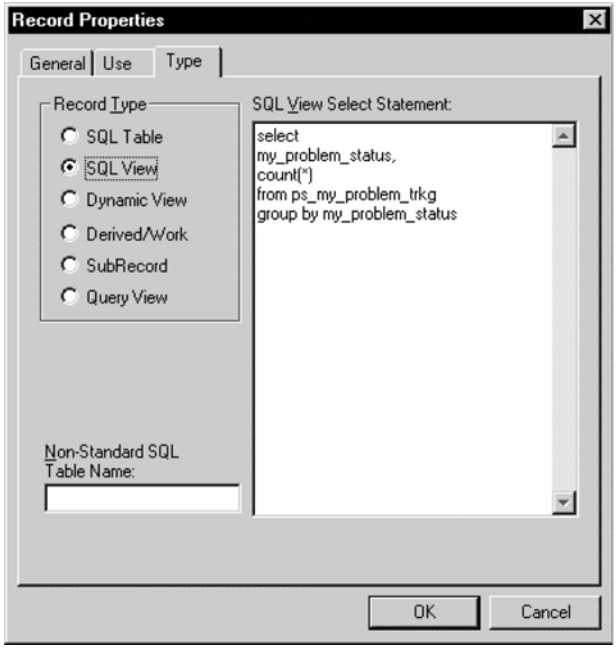


Figure A.17 MY_TRKG_STATUS View—SQL Select Statement

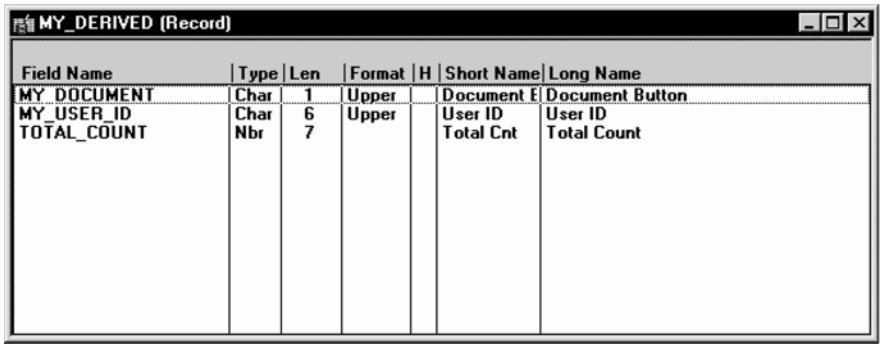


Figure A.18 MY_DERIVED Work Record—Field Display

MY_DERIVED (Record)

Field Name	Type	Key	Dir	CurC	Srch	List	Sys	Audt	H	Default
MY_DOCUMENT	Char				No	No	No			
MY_USER_ID	Char				No	No	No			
TOTAL_COUNT	Nbr				No	No	No			

Figure A.19 **MY_DERIVED Work Record—Use Display**

MY_DERIVED (Record)

Field Name	Type	Req	Edit	Prompt Table	Set Control Field	Rs Dt
MY_DOCUMENT	Char	No	Y/N			No
MY_USER_ID	Char	No				No
TOTAL_COUNT	Nbr	No				No

Figure A.20 **MY_DERIVED Work Record—Edits Display**

MY_DERIVED is a derived record that holds three fields which are used as work fields in our application.

MY_PROBLEM_TRKG (Record PeopleCode)

MY_PROBLEM_ID | FieldDefault

```
SQLExec("select max(my_problem_id) from ps_my_problem_trkg",
&MAX_PROBLEM_ID);
If All(&MAX_PROBLEM_ID) Then
   &NEXT_PROBLEM_ID = LTrim(&MAX_PROBLEM_ID, "0");
   &NEXT_PROBLEM_ID = &MAX_PROBLEM_ID;
   &NEXT_PROBLEM_ID = &NEXT_PROBLEM_ID + 1;
   &NEXT_ID_STRING = String(&NEXT_PROBLEM_ID);
   &ID_LENGTH = Len(&NEXT_ID_STRING);
   &ID_DIFF = 6 - &ID_LENGTH;
   If &ID_DIFF > 0 Then
      MY_PROBLEM_ID = Rept("0", &ID_DIFF) | &NEXT_ID_STRING;
   Else
      MY_PROBLEM_ID = &NEXT_ID_STRING;
   End-If;
Else
   MY_PROBLEM_ID = "000001";
End-If;
```

Figure A.21 MY_PROBLEM_TRKG.MY_PROBLEM_ID.**FieldDefault**

MY_PROBLEM_TRKG (Record PeopleCode)

CLOSE_DT | SaveEdit

```
If (MY_PROBLEM_STATUS = "5" Or
      MY_PROBLEM_STATUS = "6") And
      None(CLOSE_DT) Then
   Warning ("All incidents that are resolved/voided must have a close
date");
End-If;
```

Figure A.22 MY_PROBLEM_TRKG.CLOSE_DT.**SaveEdit**

The first PeopleCode program automatically increments the MY_PROBLEM_ID field to the next one. The second PeopleCode program performs an edit to ensure that the CLOSE_DT field is entered when incidents are resolved or voided.

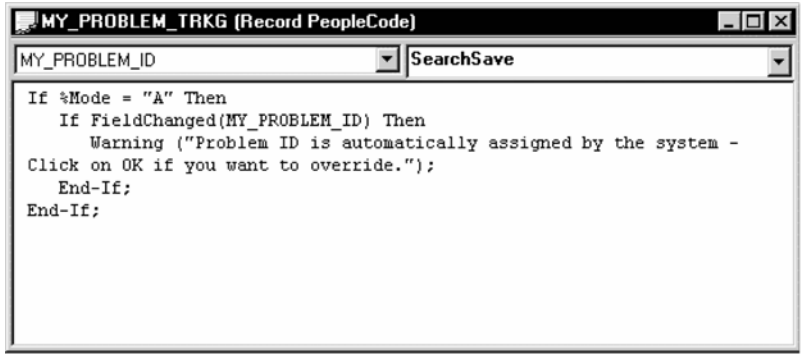


Figure A.23 MY_PROBLEM_TRKG.MY_PROBLEM_ID.**SearchSave**

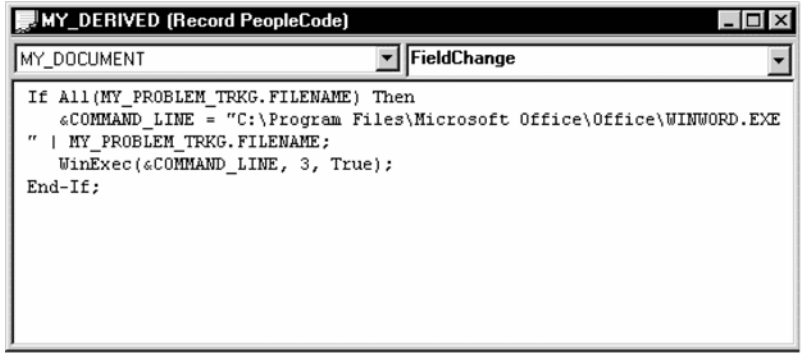


Figure A.24 MY_DERIVED.MY_DOCUMENT.**FieldChange**

The first PeopleCode program prevents the user from assigning a value to the MY_PROBLEM_ID field when new incidents are added using our application. The second PeopleCode program opens a Microsoft Word document explaining an incident in our application.

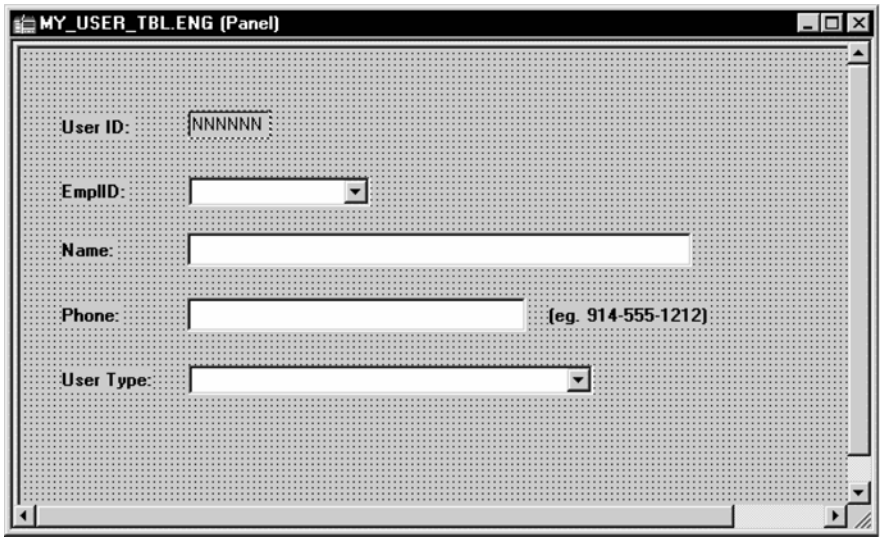


Figure A.25 **MY_USER_TBL panel**

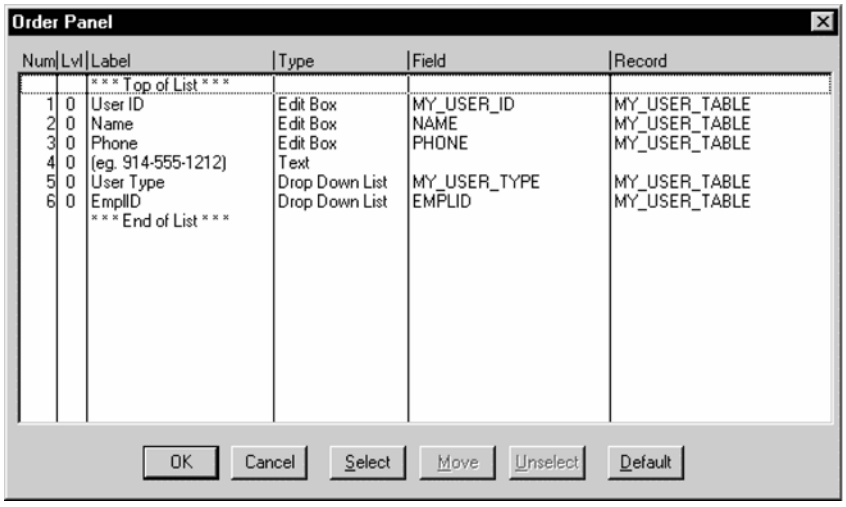


Figure A.26 **MY_USER_TBL panel layout**

MY_USER_TBL panel is used to enter user information on our application.

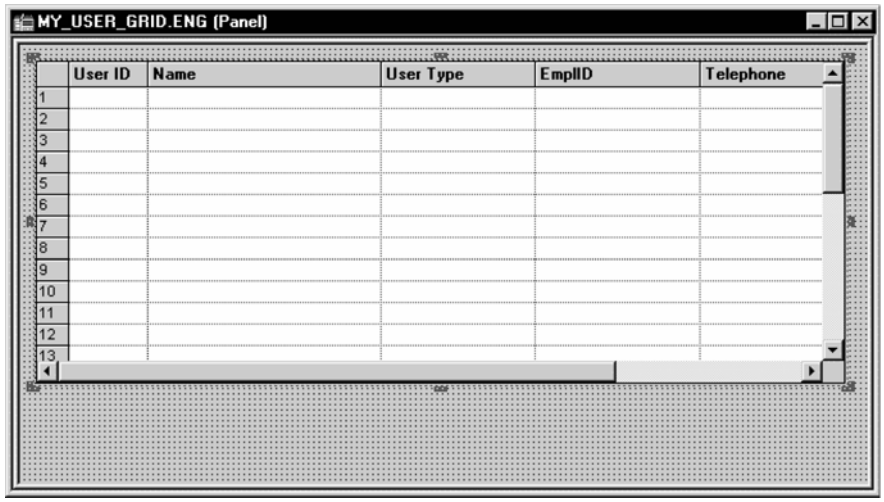


Figure A.27 **MY_USER_GRID panel**

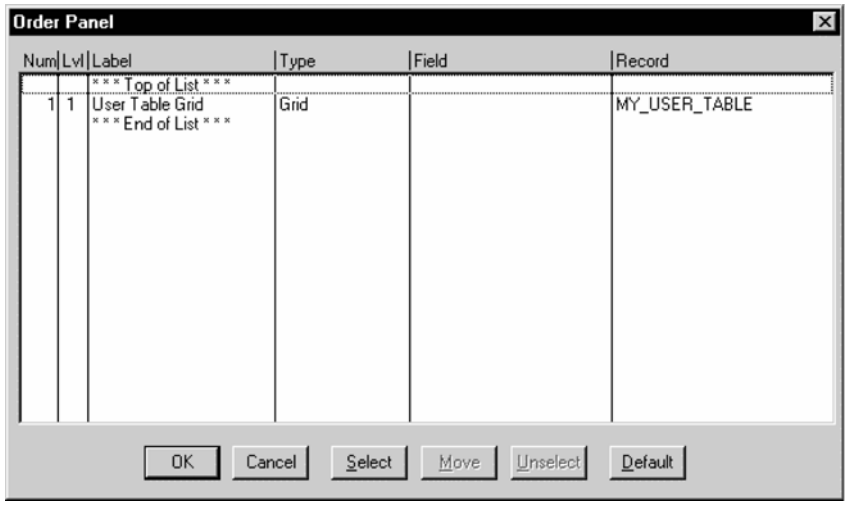


Figure A.28 **MY_USER_GRID panel layout**

MY_USER_GRID panel is used to enter user information in a grid format.

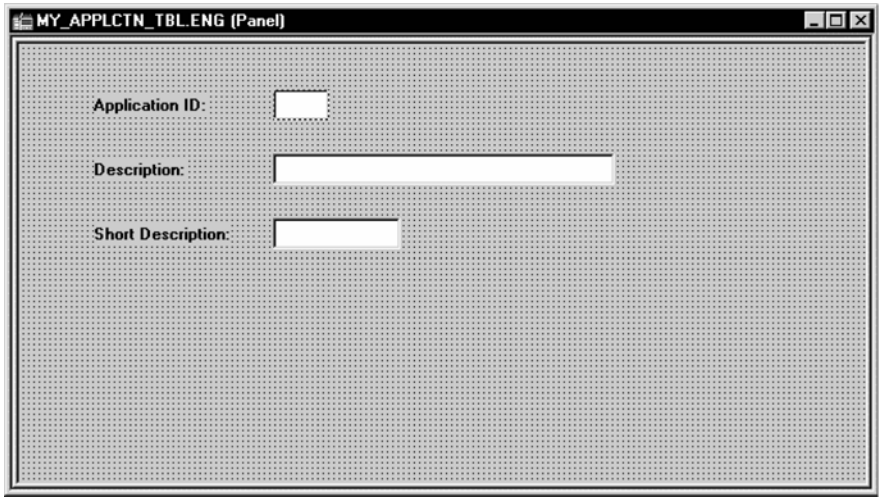


Figure A.29 **MY_APPLCTN_TBL panel**

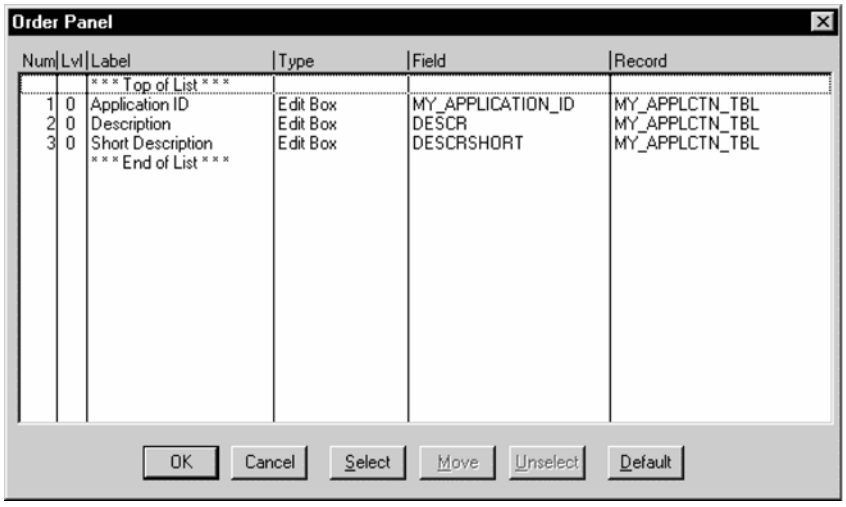


Figure A.30 **MY_APPLCTN_TBL panel layout**

MY_APPLCTN_TBL panel is used to set up applications that are tracked through our Problem Tracking application.

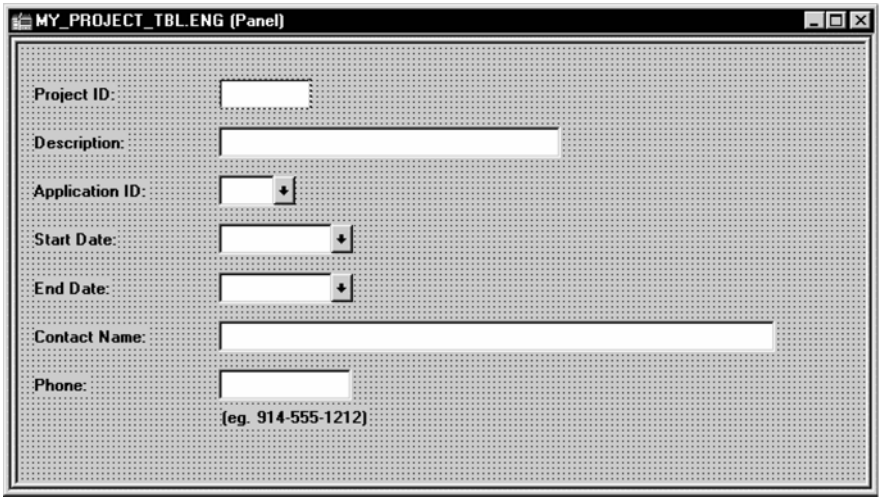


Figure A.31 **MY_PROJECT_TBL panel**

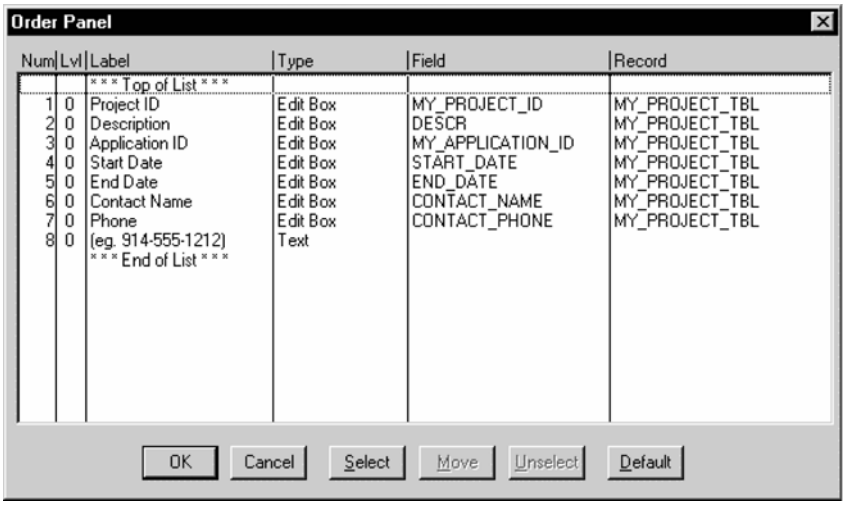


Figure A.32 **MY_PROJECT_TBL panel layout**

MY_PROJECT_TBL panel is used to enter project information in our application.

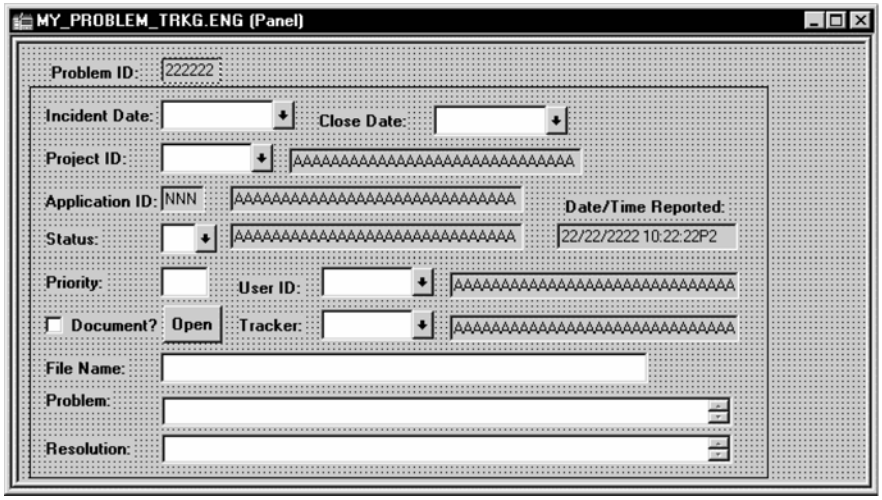


Figure A.33 MY_PROBLEM_TRKG panel

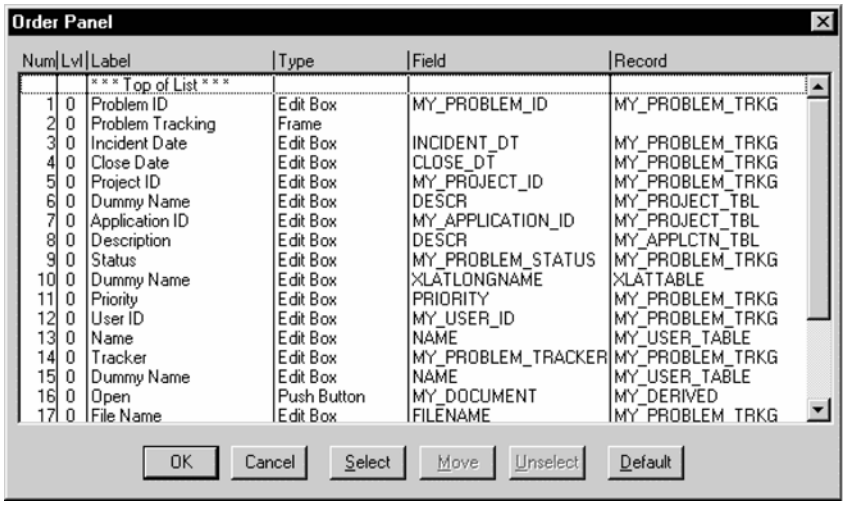


Figure A.34 MY_PROBLEM_TRKG panel layout

MY_PROBLEM_TRKG panel is used to enter incidents and resolutions through our Problem Tracking application.

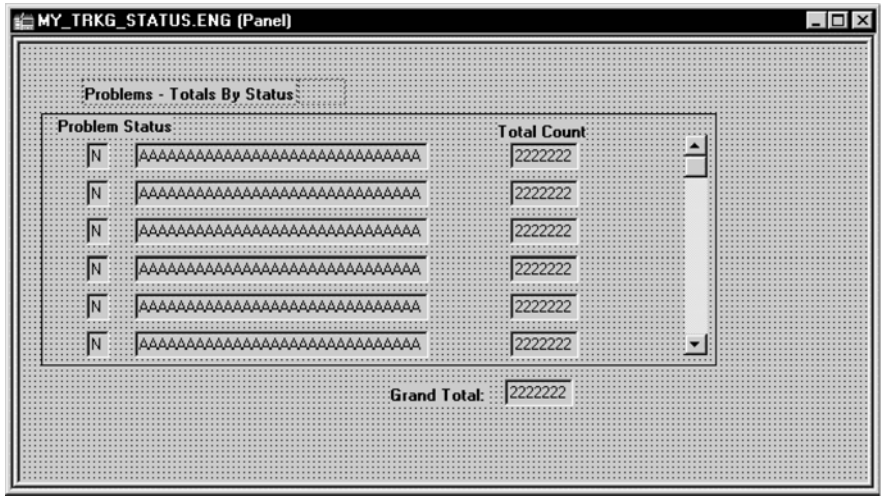


Figure A.35 **MY_TRKG_STATUS panel**

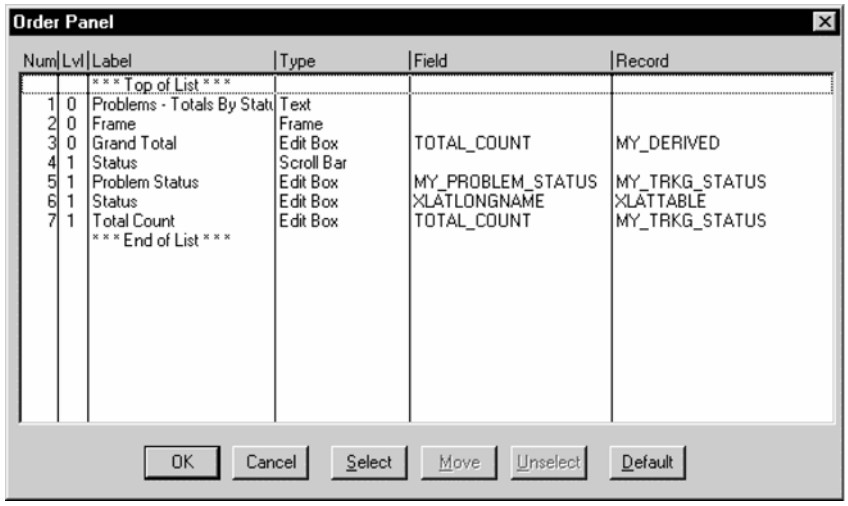


Figure A.36 **MY_TRKG_STATUS panel layout**

MY_TRKG_STATUS panel is used to view totals of all incidents/problems tracked using our application.

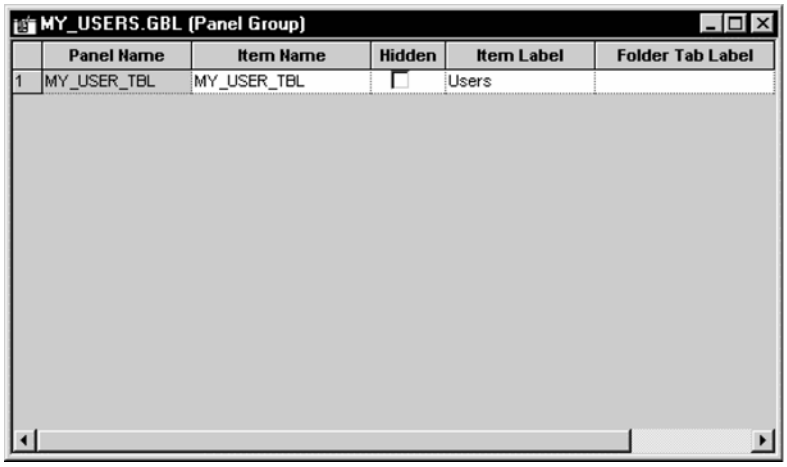


Figure A.37 MY_USERS panel group

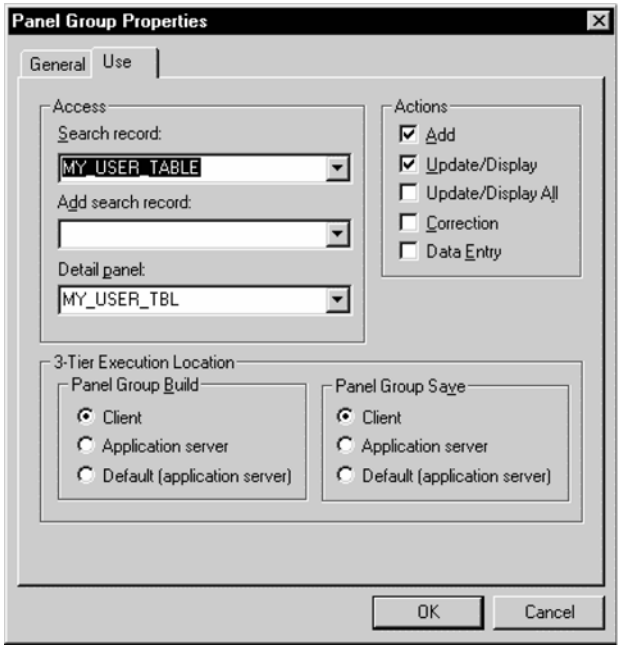


Figure A.38 MY_USERS panel group properties

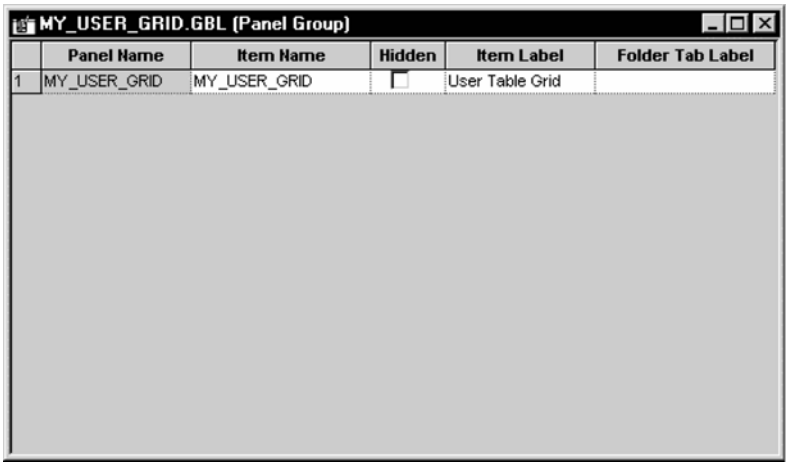


Figure A.39 **MY_USER_GRID panel group**

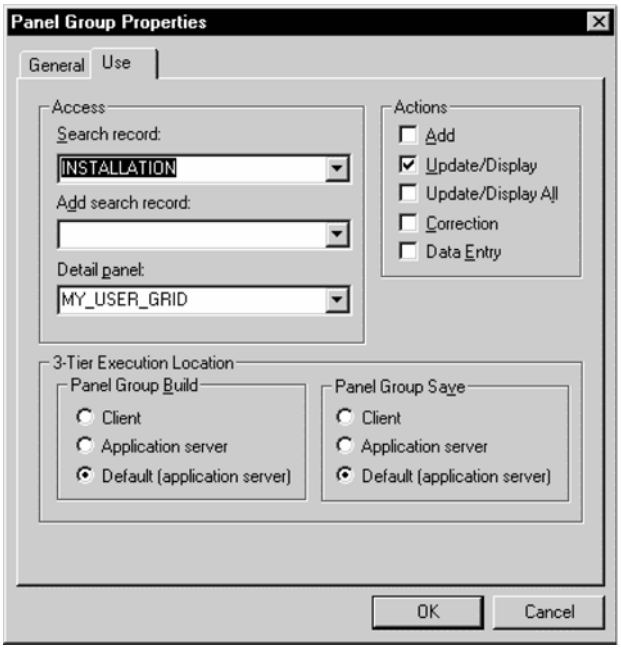


Figure A.40 **MY_USER_GRID panel group properties**

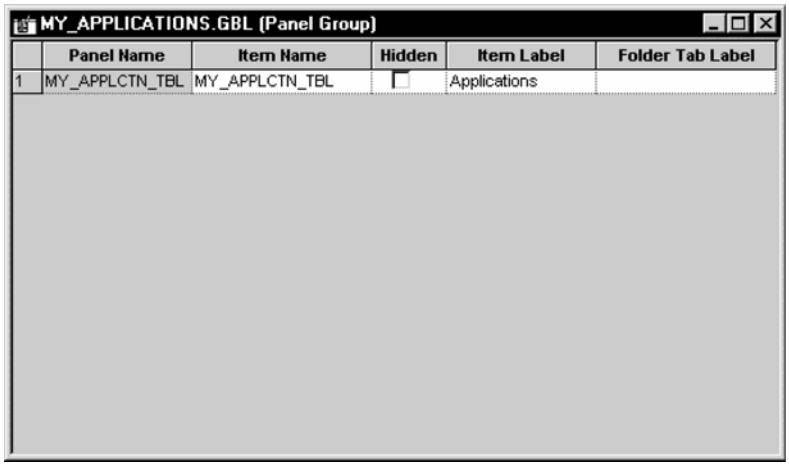


Figure A.41 **MY_APPLICATIONS panel group**

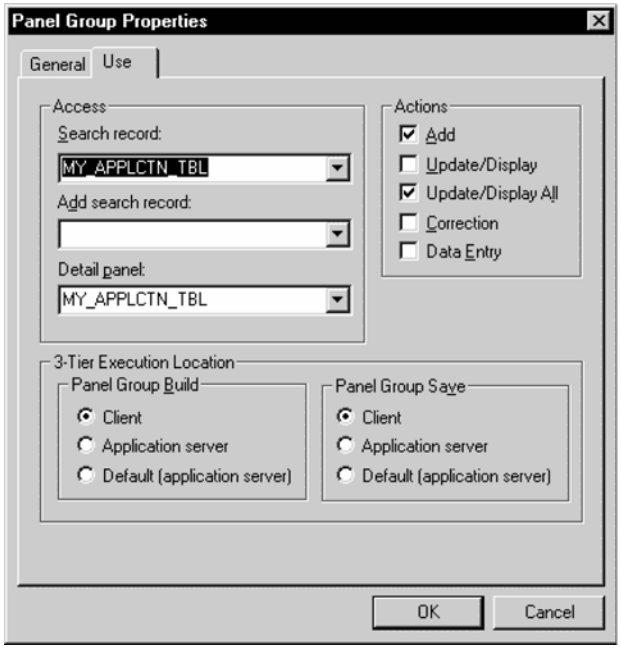


Figure A.42 **MY_APPLICATIONS panel group properties**

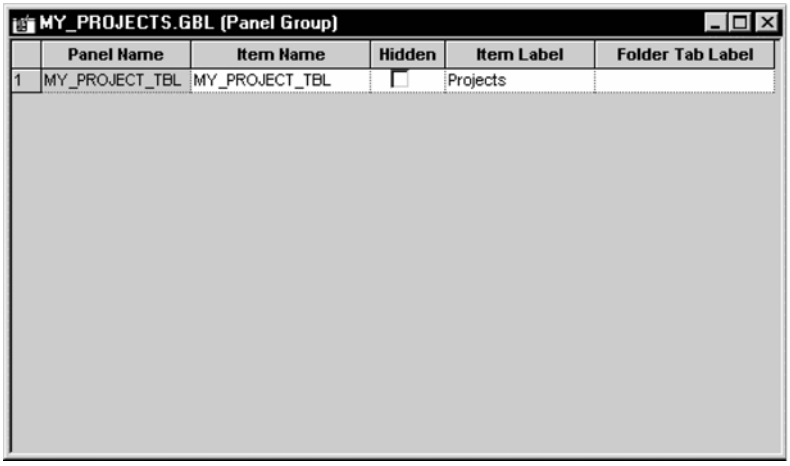


Figure A.43 **MY_PROJECTS panel group**

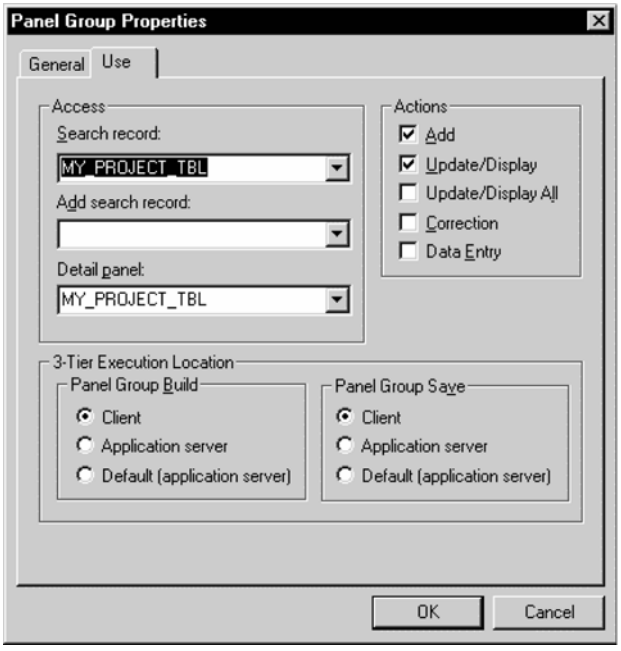


Figure A.44 **MY_PROJECTS panel group properties**

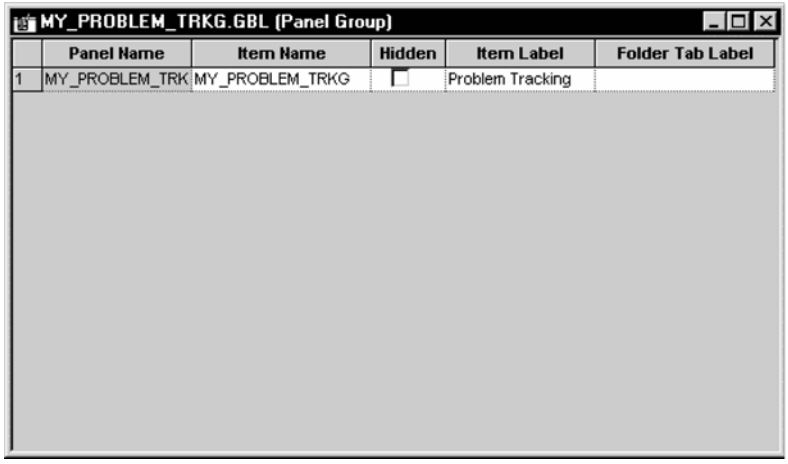


Figure A.45 **M_PROBLEM_TRKG panel group**

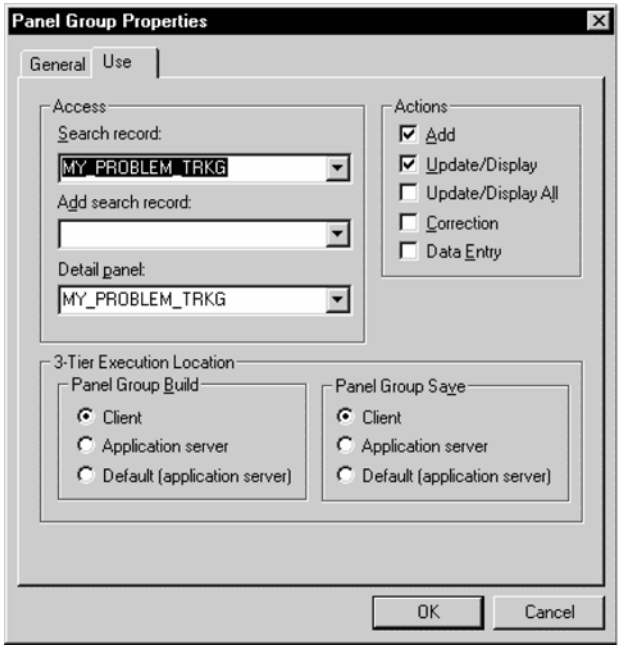


Figure A.46 **M_PROBLEM_TRKG panel group properties**

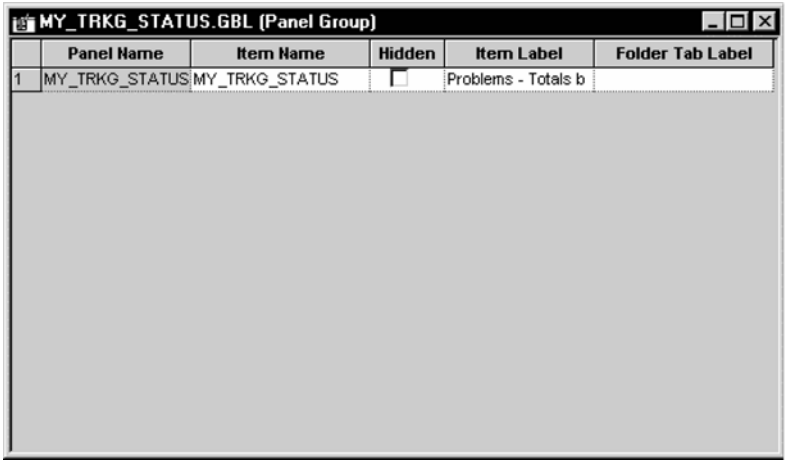


Figure A.47 **MY_TRKG_STATUS panel group**

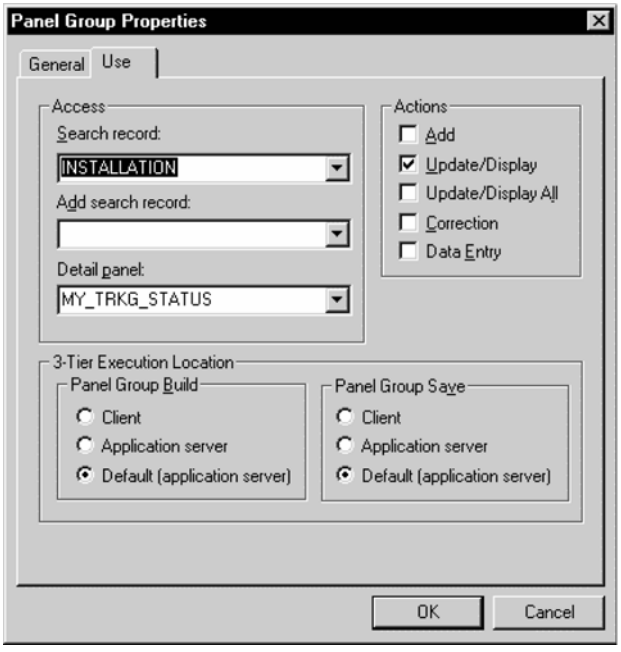


Figure A.48 **MY_TRKG_STATUS panel group properties**

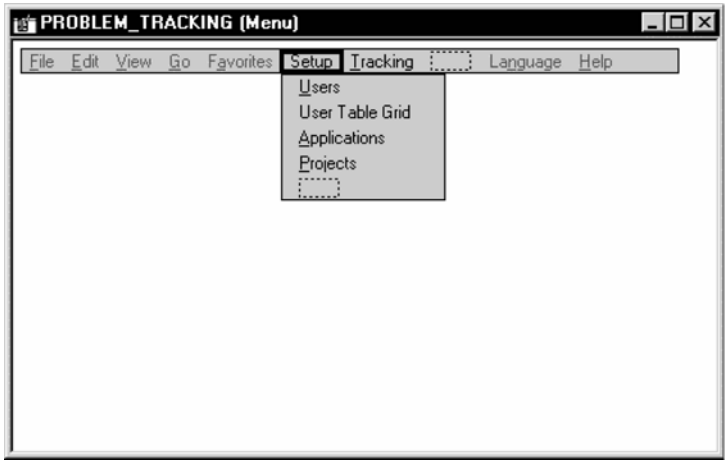


Figure A.49 Problem Tracking menu—setup bar items

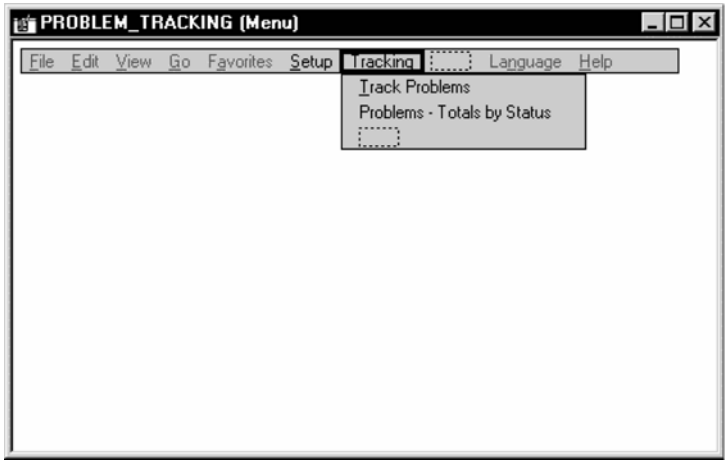


Figure A.50 Problem Tracking menu—tracking bar items

Figure A.50 Problem Tracking menu properties

A P P E N D I X B

Operator Class/Locations

The Operator Class and Employee Locations application links PeopleSoft operator classes to office locations. Employees currently working out of these office locations can be linked to the Operator Class/Location for security and reporting purposes.

The application is comprised of three main records, two views, and a Derived/Work record which stores the PeopleCode statements. The application is used primarily to demonstrate the use of scroll-related functions.

The records are:

MY_LOCATION_HDR
MY_LOCATIONS
MY_LOCATION_EMP

The two Views are:

MY_LOC_OPR_VW
MY_LOC_EMPL_VW

The Derived/Work record is:

MY_DERIVED

Figure B.1 graphically illustrated the records, the views, and the Derived/Work record for this Applet.

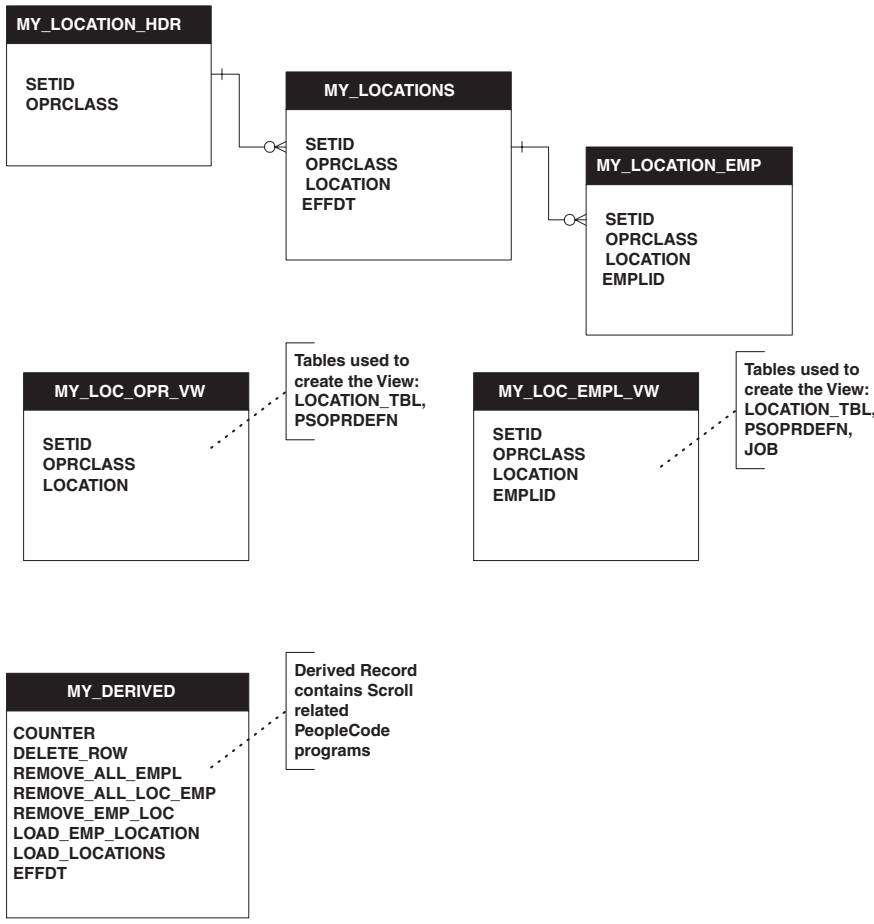


Figure B.1 Operator Classes linked to Employee Locations. Records, Views and Derived/Work

MY_LOCATION_HDR (Record)

Field Name	Type	Key	Dir	CurC	Srch	List	Sys	Audt	H	Default
SETID	Char	Key	Asc		Yes	Yes	No			OPR_DEF_TBL_HR.SETID
OPRCLASS	Char	Key	Asc		Yes	Yes	No			

Figure B.2 MY_LOCATION_HDR record

MY_LOCATIONS (Record)

Field Name	Type	Key	Dir	CurC	Srch	List	Sys	Audt	H	Default
SETID	Char	Key	Asc		Yes	Yes	No			OPR_DEF_TBL_HR.SETID
OPRCLASS	Char	Key	Asc		Yes	Yes	No			
LOCATION	Char	Key	Asc		Yes	No	No			
EFFDT	Date				No	No	No			%Date

Figure B.3 MY_LOCATIONS record

MY_LOCATION_EMP (Record)

Field Name	Type	Key	Dir	CurC	Srch	List	Sys	Audt	H	Default
SETID	Char	Key	Asc		Yes	Yes	No			OPR_DEF_TBL_HR.SETID
OPRCLASS	Char	Key	Asc		Yes	Yes	No			
LOCATION	Char	Key	Asc		Yes	No	No			
EMPLID	Char	Key	Asc		No	No	No			

Figure B.4 MY_LOCATION_EMP record

MY_LOC_OPR_VW (Record)

Field Name	Type	Key	Dir	CurC	Srch	List	Sys	Audt	H	Default
SETID	Char	Key	Asc		Yes	Yes	No			
OPRCLASS	Char	Key	Asc		Yes	Yes	No			
LOCATION	Char	Key	Asc		Yes	No	No			

Figure B.5 MY_LOC_OPR_VW view

MY_LOC_EMPL_VW (Record)

Field Name	Type	Key	Dir	CurC	Srch	List	Sys	Audt	H	Default
SETID	Char	Key	Asc		Yes	Yes	No			
OPRCLASS	Char	Key	Asc		Yes	Yes	No			
LOCATION	Char	Key	Asc		Yes	No	No			
EMPLID	Char	Key	Asc		No	No	No			

Figure B.6 MY_LOC_EMPL_VW view

The PeopleCode associated with this application resides primarily in the Derived/Work record MY_DERIVED as illustrated in figure B.7. Refer to part 3 for PeopleCode illustrations related to scroll processing.

MY_DERIVED (Record)

Field Name	Type	FDe	FEd	FCh	FFo	RIn	RIs	RDe	RSe	SEd	SPr	SPo	Srl	SrS	Wrk	PPr
COUNTER	Sign					Yes										
OPRCLASS	Char					Yes				Yes						
DELETE_ROW	Char			Yes						Yes						
REMOVE_ALL_EMPL	Char			Yes												
REMOVE_ALL_LOC_EMP	Char			Yes												
REMOVE_EMP_LOC	Char			Yes												
LOAD_EMP_LOCATION	Char			Yes												
LOAD_LOCATIONS	Char			Yes												Yes
EFFDT	Date			Yes												

Figure B.7 MY_DERIVED Derived/Work record

Panels used include those shown in figure B.8 through B.16.

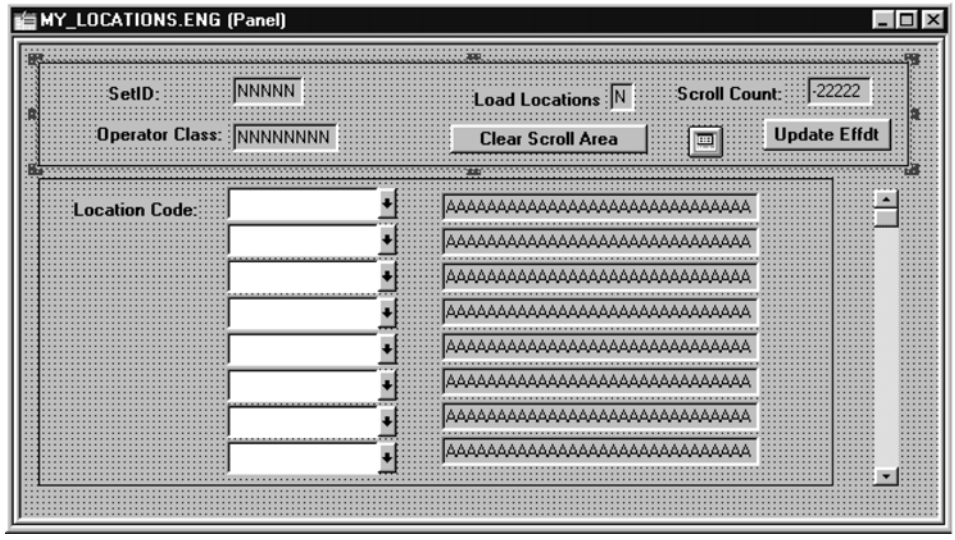
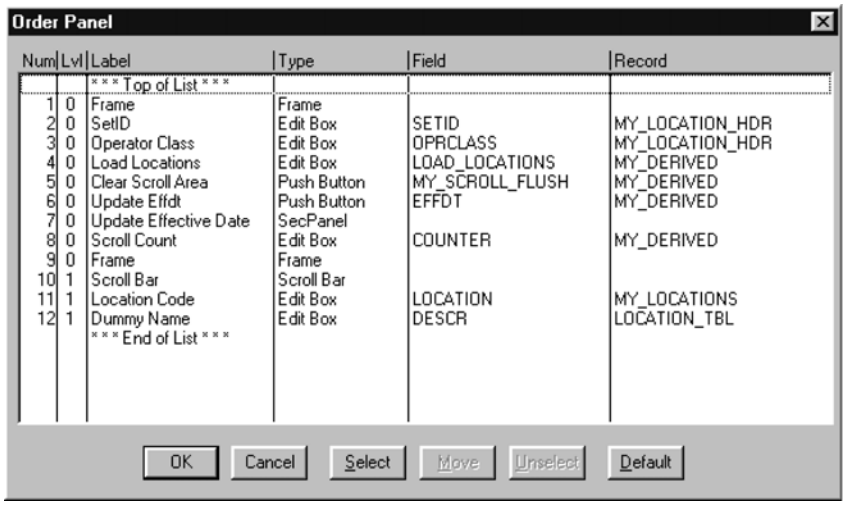


Figure B.8 MY_LOCATIONS Panel



Order Panel

Num	Lvl	Label	Type	Field	Record
		* * * Top of List * * *			
1	0	Frame	Frame		
2	0	SetID	Edit Box	SETID	MY_LOCATION_HDR
3	0	Operator Class	Edit Box	OPRCLASS	MY_LOCATION_HDR
4	0	Load Locations	Edit Box	LOAD_LOCATIONS	MY_DERIVED
5	0	Clear Scroll Area	Push Button	MY_SCROLL_FLUSH	MY_DERIVED
6	0	Update Effdt	Push Button	EFFDT	MY_DERIVED
7	0	Update Effective Date	SecPanel		
8	0	Scroll Count	Edit Box	COUNTER	MY_DERIVED
9	0	Frame	Frame		
10	1	Scroll Bar	Scroll Bar		
11	1	Location Code	Edit Box	LOCATION	MY_LOCATIONS
12	1	Dummy Name	Edit Box	DESCR	LOCATION_TBL
		* * * End of List * * *			

OK Cancel Select Move Unselect Default

Figure B.9 **MY_LOCATIONS order of Panel**

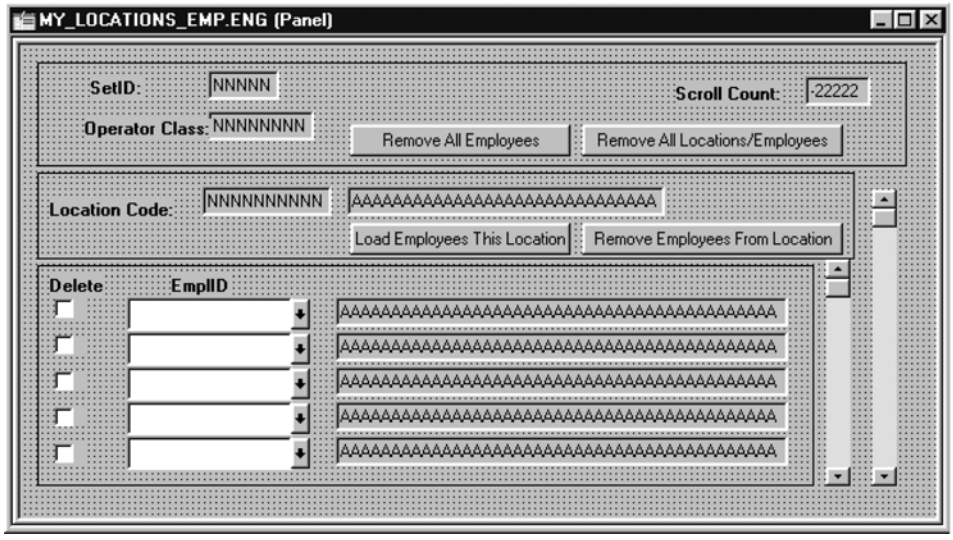
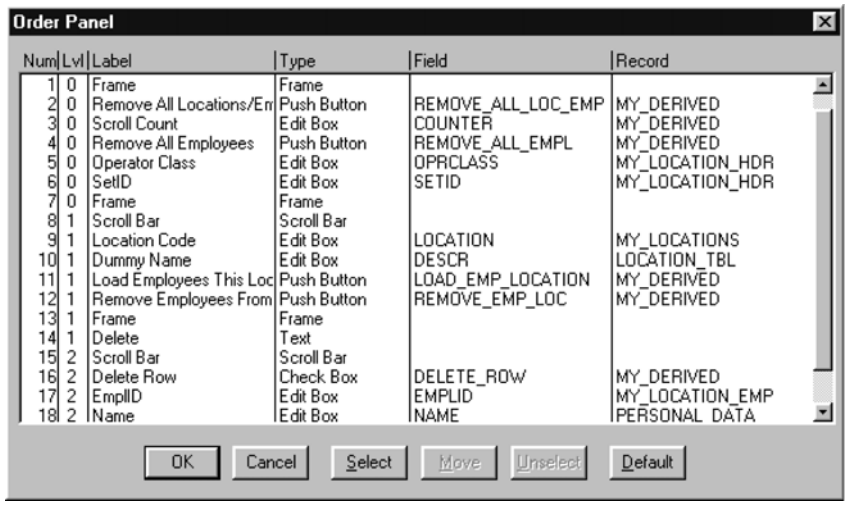


Figure B.10 **MY_LOCATIONS_EMP Panel**

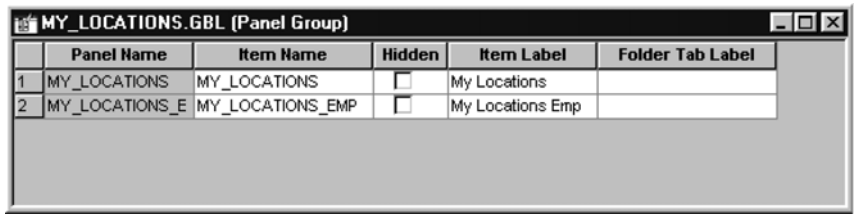


Order Panel

Num	Lvl	Label	Type	Field	Record
1	0	Frame	Frame		
2	0	Remove All Locations/Em	Push Button	REMOVE_ALL_LOC_EMP	MY_DERIVED
3	0	Scroll Count	Edit Box	COUNTER	MY_DERIVED
4	0	Remove All Employees	Push Button	REMOVE_ALL_EMPL	MY_DERIVED
5	0	Operator Class	Edit Box	OPRCLASS	MY_LOCATION_HDR
6	0	SetID	Edit Box	SETID	MY_LOCATION_HDR
7	0	Frame	Frame		
8	1	Scroll Bar	Scroll Bar		
9	1	Location Code	Edit Box	LOCATION	MY_LOCATIONS
10	1	Dummy Name	Edit Box	DESCR	LOCATION_TBL
11	1	Load Employees This Loc	Push Button	LOAD_EMP_LOCATION	MY_DERIVED
12	1	Remove Employees From	Push Button	REMOVE_EMP_LOC	MY_DERIVED
13	1	Frame	Frame		
14	1	Delete	Text		
15	2	Scroll Bar	Scroll Bar		
16	2	Delete Row	Check Box	DELETE_ROW	MY_DERIVED
17	2	EmplID	Edit Box	EMPLID	MY_LOCATION_EMP
18	2	Name	Edit Box	NAME	PERSONAL_DATA

OK Cancel Select Move Unselect Default

Figure B.11 MY_LOCATIONS_EMP Order of Panel



MY_LOCATIONS.GBL (Panel Group)

	Panel Name	Item Name	Hidden	Item Label	Folder Tab Label
1	MY_LOCATIONS	MY_LOCATIONS	☐	My Locations	
2	MY_LOCATIONS_E	MY_LOCATIONS_EMP	☐	My Locations Emp	

Figure B.12 Operator/Class & Employee Locations Panel Group

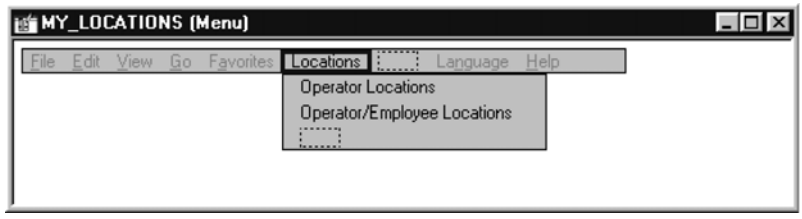


Figure B.13 Operator/Class & Employee Locations Menu

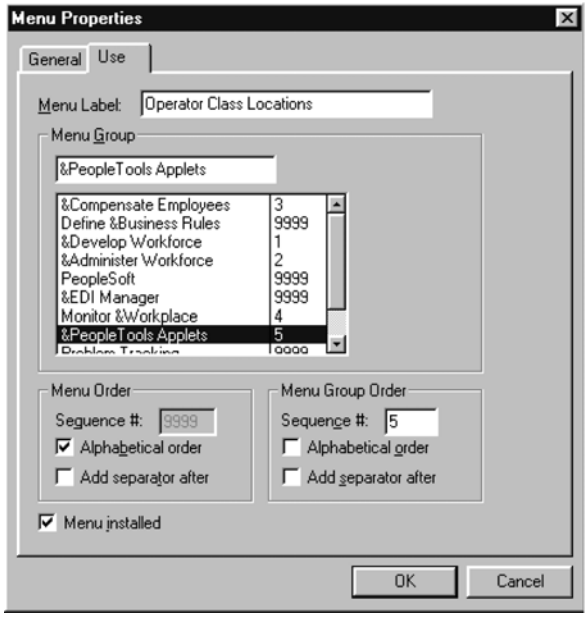


Figure B.14
Operator/Class & Employee Locations Menu Properties

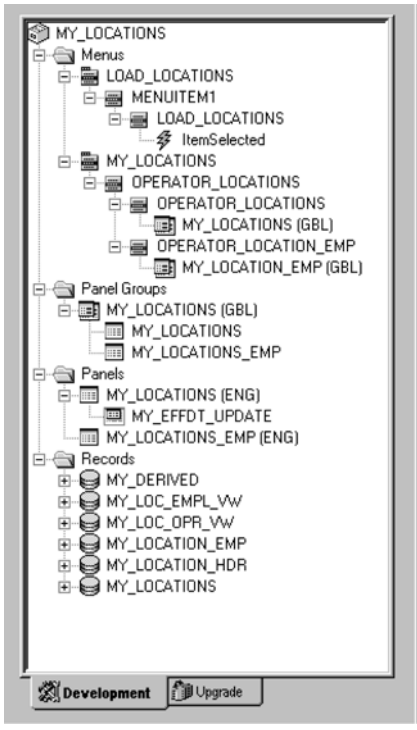


Figure B.15
Operator/Class Employee Location Project Workspace

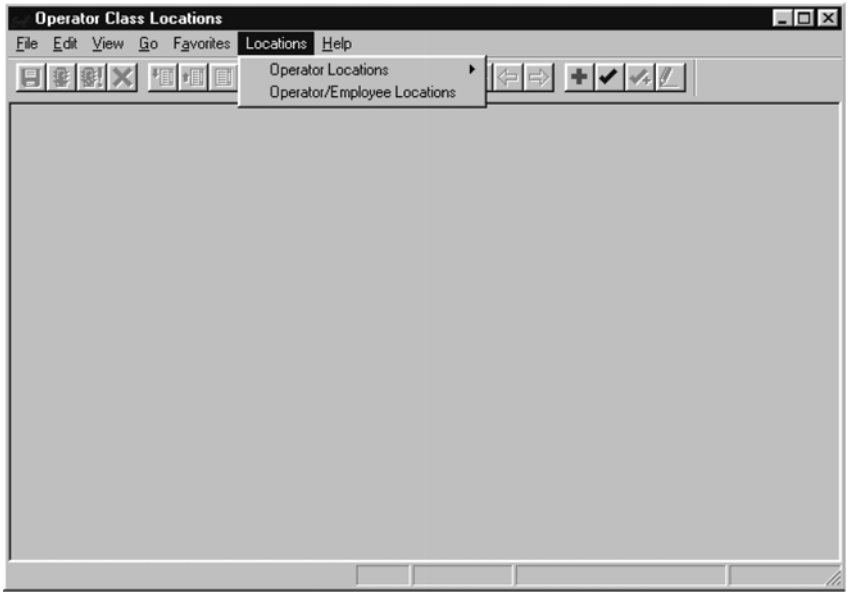


Figure B.16 Operator/Class & Employee Locations Applet as implemented

A P P E N D I X C

PeopleTool system tables

In this appendix you can find names and descriptions of the underlying PSTOOLS System tables. Some system tables have been omitted such as those that support Workflow and EDI Manager. The most common tables are listed by Tools Category.

Application Engine

AE_APPL_TBL	Application Definitions
AE_REQUEST	AE Request
AE_RUN_CONTROL	AE Run Control
AE_SECTION_TBL	Application Sections
AE_STEP_TBL	Section Steps
AE_STMT_B_TBL	AE Statement Chunk Table
AE_STMT_TBL	AE Statement Table

Change Control

PSCHGCTLHIST	Change Control History Table
PSCHGCTLLOCK	Change Control Locked Objects

Field Definition

PSDBFIELD	Database Field

Record Definition

PSDDLDEFPARMS	DDL Model Parameter

PSDDLMODEL	DDL Model Statement
PSIDXDDLPARM	Index DDL Parameters
PSINDEXDEFN	Index Definition
PSKEYDEFN	Key in Index Definition
PSPROGNAME	Record Field PeopleCode
PSRECDDLPARM	Record DDL Parameter
PSRECDEFN	Record Definition
PSRECFIELD	Record Field
PSSPCDDLPARM	Space DDL Parameters
PSVIEWTEXT	SQL View Text
XLATTABLE	Translate Value

Panel Definition

PSCOLORDEFN	Color Definition
PSPNLDEFN	Panel Definition
PSPNLFIELD	Panel Field
PSPNLTREECTRL	Panel Tree Control
PSSTYLEDEFN	Style Definition
PSTOOLBARDEFN	Toolbar Definition
PSTOOLBARITEM	Toolbar Item

Panel Group

PSPNLGROUP	Panel Group
PSPNLGRPDEFN	Panel Group Definition

Menu Definition

PSMENUDEFN	Menu Definition
PSMENUITEM	Menu Item
PSXFERITEM	Pop-up Menu Item Transfer Defns

Operator Definition

ACCESS_GRP_TBL	Tree Access Groups
PSACCESSPRFL	Access Profile
PSAUTHITEM	Authorized Menu Item
PSAUTHPRCS	Authorized Process
PSAUTHSIGNON	Authorized Signon Period
PSOBJGROUP	Object Group
PSOPRALIAS	Operator Alias
PSOPRALIASTYPE	Operator Alias Types
PSOPRCLS	Operator classes per operator
PSOPRDEFN	Operator Definition

PSOPROBJ	Operator Object Group
PSPRCSPRFL	Process Profile
PSPRCSRUNCNTL	Process Run Control
SCRTY_ACC_GRP	Access Group Security
SCRTY_QUERY	PS/Query Profile

Tree Definition

PSTREEDEFN	Tree Definition
PSTREELEAF	Tree Leaf
PSTREELEVEL	Tree Level
PSTREENODE	Tree Node
PSTREESTRCT	Tree Structure
TREE_LEVEL_TBL	Sample/Default Tree Level Tbl
TREE_NODE_TBL	Tree Nodes

Query Definition

PSQRYBIND	Query Prompt
PSQRYCRITERIA	Query Criteria
PSQRYDEFN	Query Definition
PSQRYDEL	Query Definition
PSQRYEXPR	Query Expression
PSQRYFIELD	Query Field
PSQRYRECORD	Query Record
PSQRYSELECT	Query Select

NVision Definition

NVS_REPORT	PS/nVision Report Requests
NVS_SCOPE	PS/nVision Scope
NVS_SCOPE_FIELD	PS/nVision Scope Field
NVS_SCOPE_VALUE	PS/nVision Scope Values
PSTREESELCTL	Tree Selection Control
PSTREESELNUM	Tree Select Control Number
PSTREESELECTxx	Tree Select Work-Size (01 thru 30)

PeopleCode Definition

PSPCMNAME	PeopleCode Reference
PSPCMPROG	PeopleCode Program

Utilities (Messages/Tablesets)

MESSAGE_CATALOG	Message Catalog
MESSAGE_SET_TBL	Message Sets

REC_GROUP_REC	Record Group Records
REC_GROUP_TBL	Record Groups
SETID_TBL	TableSet IDs
SET_CNTRL_GROUP	TableSet Record Groups
SET_CNTRL_REC	TableSet Record Detail
SET_CNTRL_TBL	TableSet Controls
SET_CNTRL_TREE	TableSet Tree Controls

Import Definitions

PSIMPFIELD	Import Field
PSIMPDEFN	Import Definition

Upgrader Definition

PSOBJCHNG	Object Change
PSPROJECTDEFN	Project Definition Table
PSPROJECTITEM	Project Item Table
PSPROJECTMSG	Project Messages
PSRELEASE	Release Table
PST_PNLFIELDS	Upgrader Panel Work

Process Scheduler

PRCSDEFN	Process Definitions
PRCSDEFNGRP	Process Definition Groups
PRCSDEFNPNL	Process Definition Panelgroups
PRCSDEFNXFER	Process Definition Transfers
PRCSJOBDEFN	Process Job Definitions
PRCSJOBGRP	Process Job Groups
PRCSJOBITEM	Process Job Items
PRCSJOBPNL	Process Job Panel Groups
PRCSRUNCNTL	Process Run Control Template
PRCSSAMPLER	Process Scheduler Example
PRCSSYSTEM	Process System Table
PRCSTYPEDEFN	Process Type Definitions
PSPRCSLOCK	Process Scheduler Lock Table
PSPRCSRQST	Process Request
PSPRCSRQSTXFER	Process Request Transfer
PSRECURDEFN	Process Recurrence Definition
PSSERVERSTAT	Process Server Statistics
SERVERCLASS	Server Classes
SERVERDEFN	Process Server Definition

COBOL Definition

MESSAGE_LOG	Message Log Table
MESSAGE_LOGPARM	Message Parameter Log
SQLSTMT_TBL	Stored SQL Statements

Mass Change

MC_DATA_TBL	Mass Change SQR Datatypes
MC_DEFN	Mass Change Definition
MC_DEFN_CRIT	Mass Change Defn Criteria
MC_DEFN_CRIT_VL	Mass Change Defn Crit Values
MC_DEFN_DEFAULT	Mass Change Defn Defaults
MC_DEFN_DESCR	Mass Change Defn Description
MC_DEFN_PT	Mass Change Defn PeopleTools
MC_DEFN_SQL	Mass Change Defn SQL
MC_DEFN_SQL_LN	Mass Change Defn SQL Line
MC_DEFN_STMNT	Mass Change Defn Statement
MC_DTTM_PARMS	Mass Change Datetime Parms
MC_GROUP	Mass Change Defn Group
MC_GROUP_LN	Mass Change Defn Group Line
MC_HIST_CRIT	Mass Change History Criteria
MC_HIST_CRIT_VL	Mass Change History Crit Value
MC_HIST_DEFAULT	Mass Change History Defaults
MC_HIST_STMNT	Mass Change History Statement
MC_OPRID	Mass Change Operator Security
MC_OPR_SECURITY	Mass Change Operator Security
MC_PROMPTS	Mass Change Prompt Table Setup
MC_RUN_CNTL	Mass Change Run Control
MC_TEMPLATE	Mass Change Template
MC_TEM_CRITERIA	Mass Change Template Criteria
MC_TEM_DEFAULTS	Mass Change Template Defaults
MC_TEM_DESCR	Mass Change Template Descr
MC_TEM_STMNT	Mass Change Template Statement
MC_TYPE	Mass Change Type
MC_TYPE_DESCR	Mass Change Type Description
MC_TYPE_FIELD	Mass Change Type Field
MC_TYPE_JOIN	Mass Change Type Join Table
MC_TYPE_RECORD	Mass Change Type Record
MC_TYPE_SQL	Mass Change Type SQL Statement
MC_TYPE_STMNT	Mass Change Type Statement
MC_TYPE_WHERE	Mass Change Type Where Clause

International Tables

COUNTRY_TBL	Countries
CURRENCY_CD_TBL	Currency Codes

System Tables

PSASOFDATE	SQR Request Dates
PSCLOCK	Database Clock Access
PSCOLORDEFN	Color Definition
PSFMTDEFN	Format Definition Table
PSFMTITEM	Format Item Table
PSLOCK	PeopleTools System Control
PSOPTIONS	PeopleTools System Options
PSSTYLEDEFN	Style Definition
RUN_CNTL_SYSAUD	SysAudit Control Table
STRINGS_TBL	Strings Table

A P P E N D I X D

Application Engine examples

In this appendix you can find the Application Engine source code used in our exercises. Each section/step is listed along with the statement type and the statement text used (if any). Any called sections appear next to the statement type (in the case of `DO Select`, `DO When`, or `DO` section types).

Exercise #1—Application USER001

Section/Step MAIN.STEP1
Statement Type `Update`
Statement Text
```
&MSG(,1,'Hello World')
```

Exercise #2—Application USER002

Section/Step MAIN.STEP1
Statement Type `Select`
Statement Text
```
&SELECT(COUNTER)
SELECT COUNT(*)
  FROM PS_PERSONAL_DATA
```

Section/Step MAIN.STEP2
Statement Type `Update`
Statement Text
```
&MSG(,2,'PERSONAL_DATA Record Count ',
  &BIND(COUNTER))
```

Exercise #3—Application USER003

Section/Step MAIN.STEP1
Statement Type Select
Statement Text

```
&SELECT(COUNTER)
SELECT COUNT(*)
  FROM PS_&BIND(RECNAME,NOQUOTES,STATIC)
```

Section/Step MAIN.STEP2
Statement Type Update
Statement Text

```
&MSG(,3,
   &BIND(RECNAME,NOQUOTES),&BIND(COUNTER))
```

Exercise #4—Application USER004

Section/Step MAIN.STEP1
Statement Type DO Select (Calls Section COUNT)
Statement Text

```
&SELECT(RECNAME)
SELECT A.RECNAME
 FROM PSRECFIELD        A,
    PSRECDEFN           B
 WHERE A.RECNAME      = B.RECNAME
  AND A.FIELDNAME     = &BIND(FIELDNAME)
  AND B.RECTYPE       = 0
 ORDER BY A.RECNAME
```

Section/Step COUNT.STEP1
Statement Type Select
Statement Text

```
&SELECT(COUNTER)
SELECT COUNT(*)
  FROM PS_&BIND(RECNAME,NOQUOTES,STATIC)
```

Section/Step COUNT.STEP2
Statement Type Update
Statement Text

```
&MSG(,3,
   &BIND(RECNAME,NOQUOTES),&BIND(COUNTER))
```

Exercise #5—Application USER005

Section/Step MAIN.STEP1
Statement Type DO Select (Calls Section COUNT)
Statement Text

```
&SELECT(RECNAME)
SELECT A.RECNAME
   FROM PSRECFIELD       A,
         PSRECDEFN       B
  WHERE A.RECNAME      = B.RECNAME
```

```
   AND A.FIELDNAME    = &BIND(FIELDNAME)
   AND B.RECTYPE      = 0
 ORDER BY A.RECNAME
```

Section/Step COUNT.STEP1
Statement Type `Select`
Statement Text

```
&SELECT(COUNTER)
SELECT COUNT(*)
 FROM PS_&BIND(RECNAME,NOQUOTES,STATIC)
```

Section/Step COUNT.STEP2
Statement Type `Do When` (Calls Section MSG)
Statement Text

```
&SELECT(AE_DECIDE)
SELECT 'X'
 FROM PSLOCK
 WHERE &BIND(COUNTER) > 0
```

Section/Step MSG.STEP1
Statement Type `Update`
Statement Text

```
&MSG(,3,
  &BIND(RECNAME,NOQUOTES),&BIND(COUNTER))
```

Exercise #6—Application USER006

Section/Step MAIN.STEP1
Statement Type `DO Section` (Calls Dynamic Section —HELLO or GOODBYE)
Statement Text No Statement Text

Section/Step HELLO.STEP1
Statement Type `Update`
Statement Text `&MSG(,1,'Hello World')`

Section/Step GOODBYE.STEP1
Statement Type `Update`
Statement Text `&MSG(,1,'Goodbye')`

Exercise #7—Application MYPRCSDL

Section/Step MAIN.STEP1
Statement Type `Select`
Statement Text

```
&SELECT(PRCSTYPE,PRCSNAME)
SELECT B.PRCSTYPE,B.PRCSNAME
  FROM PS_AE_RUN_CONTROL    A,
       PS_MY_RUN_CNTL_AE    B
 WHERE A.OPRID            = B.OPRID
```

```
   AND A.RUN_CNTL_ID       = B.RUN_CNTL_ID
 AND A.PROCESS_INSTANCE =
           &BIND(PROCESS_INSTANCE)
```

Section/Step MAIN.STEP2
Statement Type `Update`
Statement Text

```
&MSG(,2,&BIND(PRCSTYPE,NOQUOTES),
  &BIND(PRCSNAME,NOQUOTES))
```

Section/Step MAIN.STEP3
Statement Type `DO Select` (Calls Section DYNSECTN)
Statement Text

```
&SELECT(AE_SECTION,RECNAME)
SELECT 'PROCESS1, RECNAME
  FROM PSRECDEFN
 WHERE RECNAME = 'PRCSDEFN'
    OR RECNAME = 'PRCSDEFNGRP'
    OR RECNAME = 'PRCSDEFNPNL'
    OR RECNAME = 'PRCSDEFNXFER'
 UNION
SELECT 'PROCESS2', RECNAME
  FROM PSRECDEFN
 WHERE RECNAME = 'PSPRCSRQST'
    OR RECNAME = 'PSPNLFIELD'
 ORDER BY 1,2
```

Section/Step DYNSECTN.STEP1
Statement Type `DO Section` (Calls Dynamic Section —PROCESS1 or PROCESS2)
Statement Text No Statement Text

Section/Step PROCESS1.STEP1
Statement Type `Update`
Statement Text

```
&SELECT(COUNTER)
SELECT COUNT(*)
  FROM PS_&BIND(RECNAME,NOQUOTES,STATIC)
 WHERE PRCSTYPE = &BIND(PRCSTYPE)
   AND PRCSNAME = &BIND(PRCSNAME)
```

Section/Step PROCESS1.STEP2
Statement Type `DO When` (Calls Section DELETE1)
Statement Text

```
&SELECT(AE_DECIDE)
SELECT 'X'
  FROM PSLOCK
 WHERE &BIND(COUNTER) > 0
```

Section/Step PROCESS1.STEP3
Statement Type `DO Section` (Calls Section MESSAGE)
Statement Text No Statement Text

Section/Step DELETE1.STEP1
Statement Type `Update`
Statement Text
```
DELETE
 FROM PS_&BIND(RECNAME,NOQUOTES,STATIC)
WHERE PRCSTYPE = &BIND(PRCSTYPE)
 AND PRCSNAME = &BIND(PRCSNAME)
```

Section/Step PROCESS2.STEP1
Statement Type `Update`
Statement Text
```
&SELECT(COUNTER)
SELECT COUNT(*)
  FROM &BIND(RECNAME,NOQUOTES,STATIC)
 WHERE PRCSTYPE = &BIND(PRCSTYPE)
   AND PRCSNAME = &BIND(PRCSNAME)
```

Section/Step PROCESS2.STEP2
Statement Type `DO When` (Calls Section DELETE2)
Statement Text
```
&SELECT(AE_DECIDE)
SELECT 'X'
  FROM PSLOCK
 WHERE &BIND(COUNTER) > 0
```

Section/Step PROCESS2.STEP3
Statement Type `DO Section` (Calls Section MESSAGE)
Statement Text No Statement Text

Section/Step DELETE2.STEP1
Statement Type `Update`
Statement Text
```
DELETE
  FROM &BIND(RECNAME,NOQUOTES,STATIC)
 WHERE PRCSTYPE = &BIND(PRCSTYPE)
   AND PRCSNAME = &BIND(PRCSNAME)
```

Section/Step MESSAGE.STEP1
Statement Type `Update`
Statement Text
```
&MSG(,3,&BIND(RECNAME,NOQUOTES),
  &BIND(COUNTER))
```

A P P E N D I X E

Built-in functions

FREQUENTLY USED BUILT-IN FUNCTIONS

The following section lists frequently used built-in functions. Examples of these functions are also illustrated throughout the book.

PeopleCode built-in functions can be grouped into functional categories. Some of the more frequently used categories are:

- Conversion
- Date/Time
- Effective Date/Sequence
- Logical
- Math
- Message Catalog/Display
- Panel Buffer
- Panel Control
- Process Scheduler
- Save/Cancel
- Scroll functions
- SQL
- String
- Trace control
- Transfers

Conversion functions

Char

Description Converts a numeric value to a character based on the character set in use

Syntax
```
Char (n)
```

Rules Accepts one byte value only and not multiple values.

Returns Returns a string value based on the corresponding number passed to the function statement.

Example
```
&CHAR_VALUE = Char(70);
```

Code

Description Examines the first character passed in a text string and returns the corresponding numeric code

Syntax
```
Code(str)
```

Rules Double-byte characters are returned as numeric codes representing both bytes

Returns A number equal to the character set in use

Example
```
&NUMERIC_CODE = Code(&MY_STRING);
```

ConvertChar

Description Converts the characters identified in the source string to the target character code

Syntax
```
ConvertChar(source_str, source_str_category, output_str, target_char_code)
```

where:

`source_str` identifies the source string to be converted
`source_str_category` the language classification of the source string
`output_str` represents the converted string
`target_char_code` numeric value identifying the conversion character type

Rules Allows for conversion between character sets such as Japanese Hankaku Katakana, Zenkaku Katakana, and Hiragana. These character sets can also be converted to ASCII single-byte representation.

Source and target character code sets not supported by `ConvertChar` are processed by the function without alteration. The 0 and 1 characters are processed without conversion. A value of `-1` is returned when `ConvertChar` cannot determine the placement of source and target characters. A `-2` is returned when the characters in the source string can be partially converted. The characters that can be converted and the characters that cannot be converted are sent to the target string as they appear in the source string.

Returns A return code of `1` indicates the string was converted successfully and a `0` indicates the string was not converted. A `-1` indicates an unknown condition.

Example

```
&RETURN_CODE = ConvertChar(&KANJI_STRING, 5, ASCII_STRING, 0);
```

String

Description Takes a value stored in a non-string type and converts it to a string

Syntax

```
String(val)
```

Rules `String` can be used when field comparisons require the specific use of string data. Object data types cannot be converted using the `String` function.

Returns A string representation of the value passed to the function

Example

```
&MY_STRING = String(&NUMBER_FIELD);
```

Date/Time functions

Date

Description Converts a number formatted as YYYYMMDD and returns a `DATE` value

Syntax

```
Date (date_number)
```

Rules The input format must be a number.

Returns A `DATE` data type value

Example

```
&DATE_FIELD = Date(20000101);
```

DatePart

Description Returns part of an input DateTime value

Syntax

```
DatePart(datetime_value)
```

Rules The input value is a `DateTime` data type.

Returns Returns the date value portion

Example

```
&DATE_PORTION = DateTimeValue("01/01/00 06:30:25 AM");
&DATE_PORTION = DatePart(&DATE_PORTION);
```

DateValue

Description Converts a date string that is in the Windows standard date setting and returns a date type

Syntax

```
DateValue(date_str)
```

Rules When the input date value is in the YY/MM/DD format and Windows regional date setting is MM/DD/YY, the panel processor returns an invalid date function error message.

Returns Returns values based on the Windows regional date setting

Example

```
/* Date format is mm/dd/yy */
&DATE_FIELD = DateValue("01/01/00");
/* Date format is yy/mm/dd */
&DATE_FIELD = DateValue("00/12/31");
```

AddToDate

Description Accepts a date and three additional parameters representing number of years, months, and days. The specified values are added to the date.

Syntax

```
AddToDate (date, num_years, num_months, num_days)
```

Rules The function accounts for leap years and will subtract from the specified date when negative values are passed.

Returns A date representing the date passed and +/- the number of years, months and days.

```
/* Subtracts five years from current date, then adds six months and two days
*/
```

Example

```
&CALCULATED_DATE = AddToDate(%Date, -5, 6, 2);
```

Effective Date/Sequence functions

CurrEffdt

Description Identifies and returns a value representing the current effective date for the record at the current scroll level

Syntax

```
CurrEffDt(level_number)
```

Rules When a level is not specified, the effective date of the current scroll level is returned.

Returns The return value is a `Date` type. When used in a scroll area that does not contain an effective dated record in the primary scroll, the function returns a number.

Example

```
&EFFDT = CurrEffDt(2);
```

CurrEffSeq

Description Returns the effective sequence of a specified scroll area

Syntax

```
CurrEffSeq(level_num)
```

Rules When a level is not specified, the effective sequence returned is that of the current scroll level.

Returns A number data type representing the effective sequence of the scroll area

Example

```
If CurrEffSeq(CurrentLevelNumber()) <> &PREVIOUS_EFFSEQ Then
   SetDefault(JOB.ACTION);
   SetDefault(JOB.ACTION_REASON);
End-If;
```

CurrEffRowNum

Description Returns the effective row number of the current specified scroll area

Syntax

```
CurrEffRowNum(level_number)
```

Rules When a level is not specified, the effective row number returned is that of the current scroll level.

Returns A number data type representing the effective row number of the scroll area

Example

```
&ROW_NUMBER = CurrEffRowNum(2);
```

NextEffDt

Description Returns the value of the specified record field that exists in the next effective-dated row

Syntax

```
NextEffDt(record_field)
```

Rules Works only with effective dated records. When a next record does not exist, the statement is bypassed.

Returns An `Any` data type containing the field in the next effective-dated record

Example

```
   If JOB.DEPTID = NextEffdt(JOB.DEPTID) Then
         Gray(JOB.PAYGROUP);
End-If;
```

PriorEffDt

Description Works with effective-dated records and is a contrast to `NextEffDt`. It returns the contents of the field passed to the function statement that appear in the prior effective-dated row

Syntax

```
PriorEffdt(record_field)
```

Rules Works only with effective dated records. When there is no prior record, the statement is bypassed by the Application Processor.

Returns An `Any` data type containing the field from the prior effective-dated record

Example

```
If JOB.COMPRATE < PriorEffdt(JOB.COMPRATE) Then
   Error ("New Compensation Rate cannot be < previous rate");
End-If;
```

Logic functions

All

Description Determines whether one or more fields contain a value. The `All` function is useful in the `SaveEdit` PeopleCode event if we wish to verify that one or more fields have been entered.

Syntax

```
All(fieldlist)
```

Rules fieldlist represents one or more field names. They can be specified as

```
[recordname.] fieldname1 [, [recordname.] fieldname2]
```

Returns A Boolean. If all the fields contain a value, then the function returns `True`. If one or more fields do not contain a value, the function returns `False`.

Example

```
If All(MY_PROBLEM_RESOLTN, CLOSE_DT) Then
   Gray(MY_PROBLEM_STATUS);
End-If;
```

AllOrNone

Description This function is a combination of the `All` and `None` functions. The function returns a Boolean `True` when all the fields contain values or if none of the fields contain values. `False` is returned when there is a combination of fields containing values and fields that do not contain values.

Syntax

```
AllOrNone(FieldList)
```

Rules A character field containing blanks or a numeric field containing a zero is categorized as a `null` field value.

Returns `True` when all fields contain values or none of the fields contain values.

Example

```
If AllOrNone(CLOSE_DT, MY_PROBLEM_RESOLTN) Then
   &RETURN = MyAuditFunction();
End-If;
```

None

Description Verifies a character field contains blanks or a numeric field contains zero

Syntax

```
None(FieldList)
```

Rules A character field containing blanks or a numeric field containing a zero is categorized as a null field value.

Returns Returns `True` if the field or list of fields do not contain a value. A `False` is returned if one or more fields contain a value.

Example

```
If MY_PROBLEM_STATUS = "5" Then
   If None(CLOSE_DT) Then
      Error ("Close date is required for resolved issues");
   End-If;
End-If;
```

Math functions

Round

Description Rounds up the number passed to the specified number of decimal positions.

Syntax

```
Round (decimal, precision)
```

Rules The value represented by a decimal must be of a number data type.

Returns Returns a decimal number rounded up to the number of decimal positions in precision.

Example

```
   &ANNUAL_RT = Round(JOB.COMPRATE, 3);
```

Int

Description Removes the decimal positions from a number and returns an integer value.

Syntax

```
Int(decimal_number)
```

Rules Int does not round the input value; it only truncates the decimal positions.

Returns Returns a whole number value.

Example

```
   &HOURLY_RATE = 12.675;
   &NEW_RATE = Int(&HOURLY_RATE);
/*  Value of &New_rate is now 12 */
```

Truncate

Description Removes the specified number of digits from a decimal value.

Syntax

```
Truncate (decimal_number, digits)
```

Rules Does not perform any rounding. When the parameter identified by digits contains a zero, all numbers to the right of the decimal point are removed.

Returns Returns a number value.

Example

```
   &COMPENSATION_RATE = 2375.67895;
   &NEW_RATE = Truncate(&COMPENSATION_RATE, 3);
/* Value of &New_Rate is 2375.678 */
   &COMPENSATION_RATE = 2375.67895;
   &NEW_RATE = Truncate(&COMPENSATION_RATE, 0);
/* Value of &New_Rate is 2375 */
```

Message Catalog/Display functions

MessageBox

Description Creates and displays a message box window.

Syntax

```
MessageBox(style, title, message_set, message_num,
default_txt [, paramlist])
```

where

`style` Enables the message box window to be tailored with a blend of icons and push buttons.

`title` The message box title.

`message_set` message_set of the message catalog. Message sets 1 through 19,999 are reserved for PeopleSoft applications.

`message_num` The message number within the message set.

`default_txt` Text that is displayed in the message box when the cataloged message set is not available or message set is represented by zero.

`parmlist` List of parameters that are displayed in the text string. They can be represented as `%1`, `%2` and so on.

Rules The function return value can be interpreted if necessary. With the style parameter, two or more buttons can be included in the message box, but their use is limited to certain PeopleCode events. When the style parameter is left out or style contains more than one button, the function becomes `user think-time`, which

indicates the button action returns a value back to the function. As a result of awaiting a reply, the Application Processor suspends the PeopleCode program until the user clicks on one of the buttons contained in the message.

Returns A number value indicating which button was pressed. See table E.1 for a list of return value descriptions. A return value of zero indicates there was insufficient memory to construct the `MessageBox`.

Example

```
If MY_PROBLEM_STATUS = "2" Then
   MessageBox(289, "Incorrect Data", 20012, 1, "Project ID %1 is invalid",
MY_PROJECT_ID);
End-If;
```

Table E.1 Message return values

Returns	Description
0	Insufficient memory
-1	Warning
1	OK button was pressed
2	Cancel
3	Abort
4	Retry
5	Ignore
6	Yes
7	No

WinMessage

Description `WinMessage` is used to display information in a message box.

Syntax

```
WinMessage (message [, style] [, title]
```

`message` A text string displayed in the message box. When `WinMessage` is used as a debugging tool, a text string provides valuable information by including field contents as parameters. Utilizing `WinMessage` to assist while debugging does not require the use of `MsgGet` and `MsgGetText` functions.

`style` This parameter is optional.

`title` The message box title.

Rules From a debugging perspective, `WinMessage` can be used to display field contents and allow us to "inch" through PeopleCode statements if necessary.

Returns When the style parameter is passed, `WinMessage` returns a number indicating which button was pressed. See table E.1 for a list of return value descriptions.

A return value of zero indicates there is insufficient memory to construct the message box.

When the style parameter is not included, a Boolean value is returned. `True` indicates the OK button is pressed.

Example

```
/* This example does not use style or title */
WinMessage("A message with no style!");

/* This message has style and returns a value based on button pressed */
WinMessage("Close date cannot be less than the reported incident date", 289,
"Invalid Date");
```

Error

Description Is used to display an error message and stop processing of the active panel. Error works with messages stored in the Message Catalog or with a text string supplied to the function.

Syntax

```
Error (String)
```

Rules The value contained in string can be a literal text message or a message stored in the message catalog. The stored message must be retrieved using the `MsgGet` or `MsgGetText` functions. This is important when using translated text messages. `Error` terminates the PeopleCode program and prevents further statements from being executed. `Error`, however, produces varying results from one PeopleCode event to another. The events in which `Error` is commonly used include `FieldEdit` and `SaveEdit`. When executed in these events, the message is displayed and processing is halted. In `FieldEdit`, the field that contains the PeopleCode event is highlighted; in `SaveEdit`, no fields are highlighted. One manner in which to work around this in the `SaveEdit` event is to use the `SetCursorPos` function for the field, prior to calling `Error`. `RowDelete` is another PeopleCode event in which the `Error` function is sometimes used. When `Error` is called in `RowDelete`, the message is displayed and the row is not deleted.

Returns Does not return a value

Example

```
/* Implemented with a message string */
   Error ("All field values are required");

/* Used with a cataloged message */
   Error MsgGet(20010, 1, "All field values are required");
```

Warning

Description `Warning` is used to display a message. `Warning` differs from `Error` because it does not halt processing. The user is presented with OK and Explain buttons, then has the opportunity to correct or change data.

Syntax

```
Warning (String)
```

Rules `Warning` works with messages stored in the message catalog or a text string supplied to the function. The stored message must be retrieved using the `MsgGet` or `MsgGetText` function. When executed, the `Warning` statement terminates the PeopleCode program and prevents further statements from being executed. `Warning` produces varying results from one PeopleCode event to another. The events in which `Warning` is commonly used include `FieldEdit` and `SaveEdit`. When used in `FieldEdit`, the message is displayed and the field that contains the PeopleCode is highlighted. Placing `Warning` in `SaveEdit` displays the message but does not highlight fields. One manner in which to work around this in the `SaveEdit` event is to use the `SetCursorPos` function for the field prior to `Warning`. `RowDelete` is another PeopleCode event in which `Warning` is sometimes used. When `Warning` is called in `RowDelete` the message is displayed with OK and Cancel buttons. The user then has the option to delete the row by pressing OK or to back out of the delete by pressing Cancel.

Returns Does not return a value

Example

```
/* This message enables the user to continue after pressing OK */

   Warning("Incident status has been assigned");
```

MsgGet

Description `MsgGet` retrieve messages from the message catalog and, when necessary, substitutes the value of each parameter contained in the message text identified by `%1`, `%2`, `%3`.

Syntax

```
MsgGet (message_set, message_num,
  default_msg_text [, paramlist] )
```

Rules When a message set number less than 1 is passed, or if the message is not in the catalog, the default message text is substituted.

`MsgGet` is not a separate function. It is used in conjunction with `MessageBox`, `WinMessage`, `Error`, and `Warning`.

Returns Retrieves stored message and substitutes parameter in a paramlist, but does not return a value

Example

```
Warning (MsgGet(20011, 1, "Number of rows deleted is %1",
  &COUNT));
```

MsgGetText

Description `MsgGetText` retrieve messages from the Message Catalog and when necessary substitutes the value of each parameter contained in the message text identified by `%1`, `%2`, `%3`.

Syntax

```
MsgGetText (message_set, message_num,
  default_msg_text [, paramlist] )
```

Rules `MsgGetText` is almost identical to `MsgGet` except that the function displays the message without displaying a message set and message number.

Returns Retrieves stored message and substitutes parameter in a paramlist, but does not return a value

Example

```
Error (MsgGetText(20012, 1, "Data cannot be saved until all fields are
entered"));
```

Panel Buffer functions

DeleteRecord

Description Works on a level zero scroll record and is used to remove the parent and any corresponding child records from the database

Syntax

```
DeleteRecord (level_zero_recfield)
```

Rules Marks records to be deleted. During save processing, the row is deleted from the database. The `DeleteRecord` function cannot be executed from a `SavePostChange` or `WorkFlow` PeopleCode event because database updates are performed at the conclusion of the `Workflow` event.

Returns An optional Boolean value is returned following the completion of the function

Example

```
  &RETURN_VALUE = DeleteRecord(MY_PROJECT_ID);
```

FieldChanged

Description Is used to verify if one or more specified fields have been changed. A field can be changed on a panel or by a PeopleCode program.

Syntax There are two methods of implementing `FieldChanged`, and they are based on how the field is referenced. When the field is referenced in a scroll path, the syntax is

```
FieldChanged(scrollpath, target_row,
  [recordname.] fieldname)
```

When the field is referenced by context:

```
FieldChanged( [recordname.] fieldname)
```

Rules When performed from a record definition that is not the same as the record name, then the recordname prefix is required.

Returns Returns `True` when the contents of the Record.Fieldname have been changed since being retrieved from the database

Example

```
  If FieldChanged(PRIORITY) Then
       &RETURN = MyAuditFunction();
    End-If;
```

InsertRow

Description Inserts a new row of data into the scroll buffer. The operation is followed by the `RowInsert` PeopleCode event.

Syntax

```
InsertRow (scrollpath, target_row [, turbo])
```

Rules This function performs the same steps as if the F7 key were pressed. The `InsertRow` function is immediately followed by the `RowInsert` PeopleCode event. The remaining PeopleCode events then follow `RowInsert`. For effective-dated scrolls, the new row is inserted before the target row. When a non-effective-dated record is inserted, the new row is inserted after the row identified in the function. For effective-dated rows, the Effdt field is set to the current date, and the values that exist in the previous row are copied to the newly inserted row.

Returns An optional Boolean value is returned following the completion of the function.

Example

```
InsertRow(RECORD.MY_LOCATIONS,
CurrentRowNumber(), RECORD.MY_LOCATION_EMP);
```

PriorValue

Description Returns the prior value of a buffer field

Syntax

```
PriorValue(fieldname)
```

Rules To expect correct results, this function should be used in the `FieldEdit` and `FieldChange` events for the buffer field where `PriorValue` is called. When the value of a field is `'1'` during panel startup, and the value is then changed by the user to a `'2'` and then to `'3'`, the PriorValue function returns `'2'` when executed after the second change. The value will not be the initial `'1'`.

Returns Returns an `Any` data type

Example

```
If PriorValue(DESCRLONG) = " " Then
   CLOSE_DT = %Date;
End-If;
```

RecordChanged

Description Indicates whether a row has been modified since being retrieved from the database

Syntax

```
Contexual Reference:  RecordChanged(RECORD.target_recname)
```

When the PeopleCode program executing is on the same record, we can use

```
RecordChanged(recordname.fieldname)
```

Rules Can be used during save processing to identify updates based on changes made during a panel session

Returns Returns `True` if the record was changed by a user panel or changed from within a PeopleCode program

Example

```
If RecordChanged(MY_USER_TABLE.NAME) Then
   &RETURN = MyAuditFunction();
End-If;
```

RecordDeleted

Description Can be used to identify rows marked for deletion as a result of an operator F8 delete or a program `DeleteRow` function call

Syntax There are two methods of implementing the `RecordDeleted` function, and they are based on how the row is referenced. When the row is referenced in a scroll path, the syntax is

```
RecordDeleted(scrollpath, target_row)
```

When the row is referenced by context

```
RecordDeleted(RECORD.target_recordname)
```

Rules Deleted rows are removed from the buffer during save processing, which enables the `RecordDeleted` function to be used in most events up to and including `SavePostChg`.

Returns Returns a Boolean `True` when a row has been marked for deletion

Example

```
If RecordDeleted(RECORD.MY_LOCATIONS,
CurrentRowNumber(),       RECORD.MY_LOCATIONS_EMP) Then
   MY_DERIVED.COUNTER = ActiveRowCount(RECORD.MY_LOCATIONS,
   CurrentRowNumber(), RECORD.MY_LOCATION_EMP);
End-If;
```

RecordNew

Description Used during save processing to determine if a row is new to the database

Syntax Can be used in two ways based on how the row is referenced. When the row is referenced in a scroll path, the syntax is

```
RecordNew(scrollpath, target_row)
```

When the row is referenced by context

```
RecordNew(RECORD.target_recordname)
```

Rules In previous releases of PeopleCode this could be written as Record-New(Recordname.Fieldname)

Returns Returns a Boolean True when the record is new to the panel buffer.

Example

```
/* Using scrollpath */
If RecordNew(RECORD.MY_LOCATIONS, CurrentRowNumber(),
RECORD.MY_LOCATION_EMP) Then
   &RETURN = MyAuditFunction();
End-If;
/* Using contextual reference) */
If RecordNew(RECORD.MY_LOCATIONS) Then
   &RETURN = MyAuditFunction();
End-If;
```

Panel Control functions

Gray

Description Sets a field on a panel so that it is display only and cannot be changed

Syntax

```
Gray(fieldname)
```

Rules The `Gray` function is commonly used in the `RowInit` event and can appear in other events such as `FieldChange`.

Returns Returns a Boolean that can be used to determine if the function was successful

Example

```
If MY_PROBLEM_STATUS = "5" Then
  Gray(CLOSE_DT);
End-If;
```

Hide

Description Hides a field on a panel making it invisible to the user

Syntax

```
Hide (fieldname)
```

Rules `Hide` can be used in a `RowInit` event, but can also appear in events such as `FieldChange` when fields are hidden based on changes made to corresponding data elements.

Returns Returns a Boolean, which can be used to determine if the function was successful

Example

```
If MY_PROBLEM_STATUS <> "5" Then
   Hide(MY_PROBLEM_RESOLTN);
End-If;
```

UnHide

Description `UnHide` makes a panel field visible again.

Syntax

```
UnHide(fieldname)
```

Rules Fields that are hidden based on the Panel Field Properties-Use-Invisible tab are not made visible because `UnHide` has no impact on these fields.

Returns Returns a Boolean, which can be used to determine if the function was successful

Example

```
If PERSONAL_DATA.BIRTHCOUNTRY <> "USA" Then
   UnHide(PERSONAL_DATA.BIRTHSTATE);
End-If;
```

Ungray

Description Allows a previously non-editable field to be editable

Syntax

```
Ungray (fieldname)
```

Rules Used in events such as `RowInit`. Can also appear in `FieldChange` after the status of a field is impacted by changes to its value or other corresponding fields.

Returns Returns a Boolean, which can be used to determine if the function was successful

Example

```
If COMPANY = DEPT_TBL.COMPANY Then
   Hide(COMPANY, PAYGROUP);
End-If;
```

Process Scheduler functions

SchedulеProcess

Description The `ScheduleProcess` function stores a row in the Process Request table enabling the system to schedule a process or job.

Syntax

```
ScheduleProcess(process_type, process_name
[, run_location] [, run_cntl_id] [, process_instance]
[, run_dttm] [, recurrence_name] [, server_name])
```

where

`process_type` A case-sensitive string that identifies the type of process to be run. SQR Report and Application Engine are examples of process types.

`process_name` An eight-character string used to identify the process

`run_location` A one character string that identifies if the process is run on the client (`'1'`) or the server (`'2'`)

`run_cntl_id` Identifies the Run Control ID that links operator IDs to Run Controls

`process_instance` The `ScheduleProcess` function receives this as a variable and assigns a unique number to identify each process requested.

`run_dttm` A process or job can be scheduled for some future time by passing a `DateTime` value in this parameter. The `%DateTime` system variable can also be passed for immediate scheduling.

`recurrence_name` Identifies the name of a recurring job or process

`server_name` Identifies the server on which the process or job will be run

Rules process_type and process_name are the only required parameters necessary to schedule a process.

When a call to `ScheduleProcess` is made from a program running on an application server, the run_location parameter cannot be passed as `'1'` (client). Doing so generates an error and subsequent cancellation of the request.

Any process involving COBOL or SQR scheduled from a program on an application server must also be run on the server. When the PeopleCode program containing `ScheduleProcess` is run on a client, the COBOL or SQR process is not restricted and can run on either the client or server.

The parameter list can accept strings in the form of bind variables or a MetaSQL string.

Returns A successful process returns zero. A non-zero return indicates an error was encountered.

Example

```
If ScheduleProcess("SQR Report", &REPORT_NAME,
   &RUN_LOCATION, &RUN_CNTL_ID) = 0 Then
   WinMessage("SQR Report successfully scheduled");
End-If;
```

Save/Cancel functions

DoCancel

Description Used to cancel activity on a panel. The function mimics the ESC key and the Cancel toolbar icon.

Syntax

```
DoCancel( )
```

Rules For the current panel group, all PeopleCode programs are terminated except for those executing in the following events:

`SaveEdit`
`SavePreChg`
`SavePostChg`

Returns Does not return a value

Example

```
If &RETURN_CODE <> 0 Then
    DoCancel();
End-If;
```

DoSave

Description Performs save processing at the conclusion of the current PeopleCode program in the `FieldEdit`, `FieldChange`, and `MenuItemSelect` events.

Syntax

```
DoSave ( )
```

Rules PeopleCode programs containing `DoSave` continue processing until the remaining statements are executed. The panel is saved at the conclusion of the program. Save processing includes the following events:

```
SaveEdit
SavePreChg
SavePostChg
WorkFlow
```

Returns Does not return a value

Example

```
If &RETURN_CODE = 0 Then
    DoSave();
End-If;
```

DoSaveNow

Description Works similar to `DoSave`, however, the panel is immediately saved without waiting for the PeopleCode program to conclude.

Syntax

```
DoSaveNow ( )
```

Rules After the panel is saved, any remaining PeopleCode statements that follow `DoSaveNow` are executed.

`DoSaveNow` is only valid from the `FieldEdit` and `FieldChange` events.

A common use of `DoSaveNow` is when remote calls are involved. When using `RemoteCall`, `DoSaveNow` can be used to save information to the database before a remote process is called.

Returns Does not return a value

Example

```
If &RETURN_CODE = 0 Then
   DoSaveNow();
   If ScheduleProcess("SQR Report", &REPORT_NAME,
&RUN_LOCATION, &RUN_CNTL_ID) = 0 Then
      &RETURN = MyAuditFunction();
   End-If;
End-If;
```

WinEscape

Description Used to cancel activity on a panel. `WinEscape` mimics the ESC key.

Syntax

```
WinEscape ( )
```

Rules Changes made to the panel since the previous save are revoked

Returns An optional Boolean value is returned if required.

Example

```
/* This example cancels the panel when fields are missing */
If None(MY_PROBLEM_STATUS, PRIORITY, MY_USER_ID) Then
   WinEscape();
End-If;
```

Scroll functions

ScrollSelect

Description Selects records from a table and loads them into the scroll buffer area of a panel. Inserts child rows under the next higher level row.

Syntax

```
ScrollSelect (levelnum, [RECORD.level1_recname,
[RECORD.level2_recname,]] RECORD.target_recname,
RECORD.sel_recname
[, sqlstr [, bindvars]]
[, turbo])
```

where

`levelnum` The level number of the target scroll area. This value can be 1, 2, or 3.

`RECORD.level1_recname` Represents the path to the target scroll area. When the target record is on scroll level 2, this parameter must precede target_recordname.

`RECORD.level2_recname` Represents the path to the target scroll area. When the target record is on scroll level 3, the target_recordname must be preceded by RECORD.level1_recname and RECORD.level2_recname.

`RECORD.target recordname` The target scroll area where the selected data are loaded. When the target scroll is on level 3, specify the level 1 and level 2 records first followed by the level 3 target scroll.

`RECORD.sel_recordname` Specifies the record or view to retrieve data from. The sel_recordname can be the same as target_recordname. One characteristic of this parameter is that it enables target rows to be loaded into a buffer with only those fields used in the scroll area, in addition to key fields. When selecting rows from a large table such as JOB (in HRMS) and the target scroll area only uses five fields, specifying a smaller target reduces the amount of data loaded into system buffers.

`sqlstr [, bindvars]` The optional SQL string parameter can contain an SQL `WHERE` and `ORDER BY` clause. One or both can be specified. The `WHERE` clause enables us to limit the number of rows loaded into the scroll area. The `ORDER BY` clause can be used to sort the rows before being loaded into the target scroll area. The SQL string can accept bind variables that are used as part of the `WHERE` or `ORDER BY` clause. Bind variables can be regular bind or inline bind variables. SQL string can include Meta-SQL functions.

`turbo` When specified, improves performance of the `ScrollSelect` function. The parameter is passed as a Boolean `True`.

Rules Allows for the specification of the target scroll area, a source record from which to select rows, and an optional SQL string. Keys on the select record must be the same as on the target scroll record. A record can be used as both select and scroll record. Select record must be defined and created using Application Designer. Select record cannot be a Derived/Work record.

Returns Does not return a value

Example

```
/* Selects data into level 2 using a target record with limited fields */
  ScrollSelect(2, RECORD.MY_LOCATIONS,
    RECORD.MY_LOCATION_EMP, RECORD.MY_LOC_EMPL_VW, True);
```

ScrollSelectNew

Description Resembles `ScrollSelect`, except that `ScrollSelectNew` marks records as new when they are loaded into the scroll area. During save processing, these records are automatically added to the database.

Syntax

```
ScrollSelectNew (levelnum,
[RECORD.level1_recname, [RECORD.level2_recname,]]
RECORD.target_recname, RECORD.sel_recname
[, sqlstr [, bindvars]]
[, turbo])
```

where

`levelnum` Represents the level number of the target scroll area. This value can be 1, 2, or 3.

`RECORD.level1_recname` Represents the path to the target scroll area. When the target record is on scroll level 2, this parameter must precede the target_recordname.

`RECORD.level2_recname` Represents the path to the target scroll area. When the target record is on scroll level 3, the target_recordname must be preceded by RECORD.level1_recname and RECORD.level2_recname.

`RECORD.target recordname` The target scroll area where the selected data are loaded. When the target scroll is on level 3, we need to specify the level 1 and level 2 records first, followed by the record at target scroll level 3.

`RECORD.sel_recordname` Specifies the record or view to retrieve data from. The sel_recordname can be the same as target_recordname. One characteristic of this parameter is that it allows target rows to be loaded into a record with only those fields used in the scroll area in addition to key fields.

`sqlstr [, bindvars]` The optional SQL string parameter can contain an SQL `WHERE` and `ORDER BY` clause. One or both can be specified. The `WHERE` clause enables us to limit the number of rows loaded into the scroll area. The `ORDER BY` clause can be used to sort the rows before they are loaded into the target scroll area. The SQL string can accept bind variables that are used as part of the `WHERE` or `ORDER BY` clause. Bind variables can be regular or inline bind variables. The SQL string can include Meta-SQL functions.

`turbo` Improves performance of the `ScrollSelectNew` function. The parameter is passed as a Boolean `True` when `Turbo` is used.

Rules Keys on the select record must be the same as on the target scroll record. A record can be used as both select and scroll record. Select record must have been defined and created using Application Designer and cannot be a Derived/Work record.

Returns Does not return a value

Example

```
/*Load Location data into scroll level 1 */
  ScrollSelectNew(1, RECORD.MY_LOCATIONS,
    RECORD.MY_LOC_OPR_VW, True);
```

ScrollFlush

Description Removes records from a target scroll area

Syntax

```
ScrollFlush (scrollpath)
ScrollPath defined as
[RECORD.level1_recname,   level1_row,]
[RECORD.level2_recname,]  level2_row,]
 RECORD.target_recname
```

where

`RECORD.level1_recname` Represents the path to the target scroll area. When the target record is on scroll level 2, this parameter must precede the target_recordname. The leve1_recname requires the RECORD prefix.

`level1_row` The level 1 row to flush. The value is an integer and can be a variable or a constant. The parameter must be specified when `ScrollFlush` is targeted at scroll levels 2 or 3.

`RECORD.level2_recname` Represents the path to the target scroll area. When the target record is on scroll level 3, the target_recordname must be preceded by a RECORD.level1_recname and RECORD.level2_recname.

`level2_row` Indicates the scroll level 2 to flush. The value is an integer and can be a variable or a constant. The parameter must be specified when `ScrollFlush` is targeted at scroll level 3.

`Target recordname` The target scroll area where rows to remove are located. When the target scroll is on level 3, specify the level 1 and level 2 records first, then the record at target level 3. The target record name must be prefixed by RECORD.

Rules Rows flushed from the target scroll area are not removed from the database

Returns Does not return a value

Example

```
ScrollFlush(RECORD.MY_LOCATIONS, CurrentRowNumber(),
 RECORD.MY_LOCATION_EMP, RECORD.MY_LOC_EMPL_VW);
```

ActiveRowCount

Description Identifies the sum of active rows in a given scroll area

Syntax

```
ActiveRowCount (Scrollpath)
ScrollPath defined as:
[RECORD.level1_recname,   level1_row,]
[RECORD.level2_recname,]  level2_row,]
 RECORD.target_recname
```

where

`RECORD.level1_recname` Represents the path to the target scroll area. When the target record is on scroll level 2 this parameter must precede the target_recordname. The leve1_recname requires the RECORD prefix.

`level1_row` Identifies the record at scroll level 1. The value is an integer and can be a variable or a constant. The parameter must be specified when `ActiveRowCount` is used to return the number of active rows at scroll level 2 or 3.

`RECORD.level2_recname` Represents the path to the target scroll area. When the target record is on scroll level 3, the target_recordname must be preceded by a RECORD.level1_recname and RECORD.level2_recname.

`level2_row` Identifies the record at scroll level 2. The value is an integer and can be a variable or a constant. The parameter must be specified when `ActiveRowCount` is used to return the number of active rows at scroll level 3.

`Target recordname` Record in the target scroll area. The target record name must be prefixed by RECORD. The target record may be on scroll level 1, 2, or 3

Rules Records marked as deleted are not included in the count.

Returns Returns a number representing the number of active rows in a scroll area

Example

```
&NUMBER_OF_ROWS = ActiveRowCount(RECORD.MY_LOCATIONS,
CurrentRowNumber(), RECORD.MY_LOCATION_EMP);
```

CurrentRowNumber

Description `CurrentRowNumber` is used when it is necessary to identify the row number of the current row in a scroll area.

Syntax

```
CurrentRowNumber ( [level] )
```

or

```
CurrentRowNumber()
```

where

`level` Identifies the scroll level where the row number is retrieved

Rules When the level parameter is not specified, the function uses the current scroll level from where the function is called as the default level. `CurrentRowNumber` is sometimes used with `ActiveRowCount` to limit program loops to the number of active rows.

Returns A number representing the current row number on the specified scroll level

Example

```
/*The return value can be used in ActiveRowCount*/
   &COUNT = ActiveRowCount(RECORD.MY_LOCATIONS, &ROW_NUMBER,
   RECORD.MY_LOCATION_EMP);

/*CurrentRowNumber can also be specified explicitlty */
   &COUNT = ActiveRowCount(RECORD.MY_LOCATIONS,
   CurrentRowNumber(1), RECORD.MY_LOCATION_EMP);
```

DeleteRow

Description `DeleteRow` enables rows to be deleted from a PeopleCode program. The function triggers the `RowDelete` event, which mimics the F8/`Delete Row` operation.

Syntax

```
DeleteRow (Scrollpath, target_row )
```

ScrollPath defined as:

```
[RECORD.level1_recname, level1_row,  ]
[RECORD.level2_recname,] level2_row,  ]
RECORD.target_recname
```

where

`RECORD.level1_recname` Represents the path to the target scroll area. When the target record is on scroll level 2, this parameter must precede the target_recordname. The leve1_recname requires the RECORD prefix.

`level1_row` Identifies the record at scroll level 1. The value is an integer and can be a variable or a constant. The parameter must be specified when `DeleteRow` is used to delete rows at scroll level 2 or 3.

`RECORD.level2_recname` Represents the path to the target scroll area. When the target record is on scroll level 3, the target_recordname must be preceded by a RECORD.level1_recname and `RECORD`.level2_recname.

`level2_row` Identifies the record at scroll level 2. The value is an integer and can be a variable or a constant. The parameter must be specified when `DeleteRow` is used to delete the number of rows at scroll level 3.

`Target recordname` The target scroll area to delete. The target record may be on scroll level 1, 2, or 3 and must be prefixed by RECORD.
`target_row` Identifies the row number to be deleted

Rules When `DeleteRow` is used in a loop, the operation must begin with the highest row and work downwards. Each time a row is deleted the system renumbers all remaining rows.

Returns Returns an optional Boolean value

Example

```
For &I = ActiveRowCount(RECORD.PERS_DATA_EFFDT)
   To 1 Step - 1
   DeleteRow(RECORD.PERS_DATA_EFFDT, &I);
End-For;
```

FetchValue

Description Retrieves the value of a field from a row stored in the panel buffer of a scroll area and places it into a variable or fieldname

Syntax

```
FetchValue (Scrollpath, target_row,
[recordname.] fieldname )
```

ScrollPath is defined as:

```
RECORD.level1_recname, level1_row,  ]
[RECORD.level2_recname,] level2_row,  ]
RECORD.target_recname
```

where

`RECORD.level1_recname` Represents the path to the target scroll area. When the target record is on scroll level 2, this parameter must precede the target_recordname. The leve1_recname requires the RECORD prefix.

`level1_row` Indicates the scroll level 1 row. The value is an integer and can be a variable or a constant. The parameter must be specified when fields from rows at level 2 or 3 are fetched.

`RECORD.level2_recname` Represents the path to the target scroll area. When the target record is on scroll level 3, the target_recordname must be preceded by a RECORD.level1_recname and RECORD.level2_recname.

`level2_row` Represents the scroll level 2 row to be referenced. The value is an integer and can be a variable or a constant. The parameter must be specified when `FetchValue` is targeted at scroll level 3.

`Target recordname` Represents the target scroll area containing the row where data are to be fetched from. The target record name must be prefixed by RECORD. The target record may be on scroll level 1, 2. or 3.

`target_row` Identifies the row number in the target scroll area where buffer field contents we will be retrieved.

`[recordname.] fieldname` The name of the field that references the value to be loaded. The record name is used when the function call is made from a record definition that is not the same as recordname. The fieldname can reside on scroll level 1, 2, or 3.

Rules In many instances `FetchValue` may not be necessary if the contents of a field are accessible to a program by using the [recordname].fieldname syntax. `FetchValue` can be used when a value is not within context.

Returns Returns an `ANY` data type value.

Example

```
&EMPLID = FetchValue(RECORD.MY_LOCATIONS,
CurrentRowNumber(), RECORD.MY_LOCATION_EMP, &I,
MY_LOCATION_EMP.EMPLID);
```

HideRow

Description HideRow is used to hide a specific row and any child rows in subordinate scroll levels.

Syntax

```
HideRow (Scrollpath)
[, target_row]
```

ScrollPath defined as:

```
[RECORD.level1_recname, level1_row,  ]
[RECORD.level2_recname,] level2_row,  ]
RECORD.target_recname
```

where

`RECORD.level1_recname` Represents the path to the target scroll area. When the target record is on scroll level 2, this parameter must precede the target_recordname. The leve1_recname requires the RECORD prefix.

`level1_row` This parameter indicates the scroll level 1 row. The value is an integer and can be a variable or a constant. The parameter must be specified when fields from rows at level 2 or 3 are to be hidden.

`RECORD.level2_recname` Represents the path to the target scroll area. When the target record is on scroll level 3, the target_recordname must be preceded by a RECORD.level1_recname and RECORD.level2_recname.

`level2_row` This parameter represents data at scroll level 2. The value is an integer and can be a variable or a constant. The parameter must be specified when `HideRow` is targeted at scroll level 3.

`Target recordname` The target record to hide. The target record may be on scroll level 1, 2, or 3 and must be prefixed by RECORD.

`target_row` Identifies the row number to be hidden

Rules When a row at a higher scroll level is hidden, any associated child rows are hidden as well. When the `HideRow` function is used, the target row is hidden but there is no impact to the underlying database tables.

Returns A Boolean indicating the success (`True`) or failure (`False`) of the call

Example

```
If MY_LOCATIONS.EFFDT < &TARGET_DATE Then
  HideRow(RECORD.MY_LOCATIONS, CurrentRowNumber(),
RECORD.MY_LOCATION_EMP, &I);
End-If;
```

HideScroll

Description This function is similar to `HideRow` except that rather than hiding a row, the complete scroll area is hidden including all data in the scroll and the scroll bar.

Syntax

```
HideScroll (Scrollpath)
```

ScrollPath is defined as:

```
[RECORD.level1_recname, level1_row,  ]
[RECORD.level2_recname,] level2_row,  ]
 RECORD.target_recname
```

where

`RECORD.level1_recname` Represents the path to the target scroll area. When the target record is on scroll level 2, this parameter must precede the target_recordname. The leve1_recname requires the RECORD prefix.

`level1_row` This parameter indicates data at scroll level one. The value is an integer and can be a variable or a constant. The parameter must be specified when hiding scroll areas at level 2 or 3.

`RECORD.level2_recname` Represents the path to the target scroll area. When the target record is on scroll level 3, the target_recordname must be preceded by a RECORD.level1_recname and RECORD.level2_recname.

`level2_row` This parameter represents data at scroll level 2. The value is an integer and can be a variable or a constant. The parameter must be specified when hiding a scroll at level 3.

`Target recordname` The target scroll area to hide. The target record may be on scroll level 1, 2, or 3 and must be prefixed by RECORD.

Rules `HideScroll` is usually implemented in the `RowInit` and `FieldChange` events.

Returns A Boolean indicating the success (`True`) or failure (`False`) of the call.

Example

```
If %Mode = "U" Then
   If ActiveRowCount(RECORD.MY_LOCATIONS) = 0 Then
      HideScroll(RECORD.MY_LOCATIONS);
   End-If;
End-If;
```

RowScrollSelect

Description `RowScrollSelect` uses the select record parameter to read data and place it into a scroll specified for a particular parent row. This function is similar to `ScrollSelect`. The difference between `ScrollSelect` and `RowScrollSelect` is that `ScrollSelect` uses the key hierarchy of the parent keys and automatically places child rows under their corresponding parent data within the scroll buffer. `RowScrollSelect` does not do this and requires that the SQL string be used to limit the rows loaded into the scroll to those of the parent row keys.

Syntax

```
RowScrollSelect (levelnum, scrollpath,
RECORD.sel_recname
[, sqlstr [, bindvars]]
[, turbo])
```

ScrollPath is defined as:

```
[RECORD.level1_recname, level1_row,
[RECORD.level2_recname, level2_row]]
RECORD.target_recname
```

where

`levelnum` Represents the level number of the target scroll area. This value can be 1, 2, or 3.

`RECORD.level1_recname` Represents the path to the target scroll area. When the target record is on scroll level 2, this parameter must precede the target_recordname. The leve1_recname requires the RECORD prefix.

`level1_row` Specifies the scroll level 1 row. The value is an integer and can be a variable or a constant. The parameter must be specified when the target record name is on scroll level 2.

`RECORD.level2_recname` Represents the path to the target scroll area. When the target record is on scroll level 3, the target_recordname must be preceded by a RECORD.level1_recname and RECORD.level2_recname.

`level2_row` This parameter represents data at scroll level 2. The value is an integer and can be a variable or a constant. The parameter must be specified when the target record name is on scroll level 3.

`target recordname` Target record name appears at the lowest scroll level. The target record name must be prefixed by RECORD. Target record may be on scroll level 1, 2, or 3. When the target record is on scroll level 2, the target record name must be prefixed with the RECORD.level1_recname, level1_row parameter. When the target is on scroll level 3, the target record name must be prefixed with the RECORD.level1_recname, level1_row and the RECORD.level2_recname, level2_row.

`RECORD.sel_recordname` Specifies the record or view from which data can be retrieved. Sel_recordname can be the same as target_recordname. One characteristic of this parameter is that it enables target rows to be loaded into a record with only those fields required in the scroll area in addition to key fields.

`sqlstr [, bindvars]` The SQL string parameter requires the SQL `WHERE` and an optional `ORDER BY` clause. The `WHERE` clause is used to limit any child keys read to those of the parent row key. The `ORDER BY` clause can be used to sort the rows before data are loaded into the target scroll area. The SQL string can accept bind variables that can be used as part of the `WHERE` or `ORDER BY` clause. Bind variables can be regular or inline bind variables. The SQL string can include Meta-SQL functions.

`turbo` Improves performance of the `RowScrollSelect` function. The parameter is passed as a Boolean `True` when `Turbo RowScrollSelect` is used.

Rules `RowScrollSelect` does not arrange child rows under their related parent row keys. It is up to the `WHERE` clause in the SQL string to limit child rows to the parent record key. Select record should be defined and created using Application Designer and cannot be a Derived/Work record.

Returns Does not return a value

Example

```
/* Loads the Direct Deposit Distribution record for the current Emplid */
   For &I = 1 To ActiveRowCount(RECORD.DIRECT_DEPOSIT);
      RowScrollSelect(2, RECORD.DIRECT_DEPOSIT, &I,
      RECORD.DIR_DEP_DISTRIB, "WHERE EMPLID = :1
      ORDER BY EFFDT", PERSONAL_DATA.EMPLID, True);
   End-For;
```

RowScrollSelectNew

Description `RowScrollSelectNew` resembles `RowScrollSelect`, except that `RowScrollSelectNew` marks records as New when they are loaded into the scroll area. RowScrollSelectNew does not automatically place child rows under their corresponding parent key within the scroll buffer. It requires that the SQL string be used to limit the rows loaded into the scroll to those of the parent key.

Syntax

```
RowScrollSelectNew (levelnum, scrollpath,
RECORD.sel_recname
[, sqlstr [, bindvars]]
[, turbo])
```

ScrollPath is defined as:

```
[RECORD.level1_recname, level1_row,
[RECORD.level2_recname, level2_row]]
RECORD.target_recname
```

where

`levelnum` Represents the level number of the target scroll area. This value can be 1, 2, or 3.

`RECORD.level1_recname` Represents the path to the target scroll area. When the target record is on scroll level 2, this parameter must precede the target_recordname. The leve1_recname requires the RECORD prefix.

`level1_row` Indicates the scroll level 1 row. The value is an integer and can be a variable or a constant. The parameter must be specified when the target record name is on scroll level 2.

`RECORD.level2_recname` Represents the path to the target scroll area. When the target record is on scroll level 3, the target_recordname must be preceded by a RECORD.level1_recname and RECORD.level2_recname.

`level2_row` Represents data at scroll level 2. The value is an integer and can be a variable or a constant. The parameter must be specified when the target record name is on scroll level 3.

`target recordname` The target record name appears at the lowest scroll level. The target record name must be prefixed by RECORD. The target record may be on scroll level 1, 2, or 3. When the target record is on scroll level 2, the target record name must be prefixed with the RECORD.level1_recname, level1_row parameter. When the target is on scroll level 3, the target record name must be prefixed with the RECORD.level1_recname, level1_row, and the RECORD.level2_recname, level2_row.

`RECORD.sel_recordname` Specifies the record or view from which data can be retrieved. sel_recordname can be the same as target_recordname. One characteristic of this parameter is that it enables target rows to be loaded into a record with only those fields required in the scroll area in addition to key fields.

`sqlstr [, bindvars]` The SQL string parameter requires the SQL `WHERE` and an optional `ORDER BY` clause. The `WHERE` clause is used to limit any child keys read to those of the parent row key. The `ORDER BY` clause can be used to sort the rows before data are loaded into the target scroll area. The SQL string can accept bind variables that can be used as part of the `WHERE` or `ORDER BY`

clause. Bind variables can be regular or inline bind variables. The SQL string can include Meta-SQL functions.

`turbo` Improves performance of the `RowScrollSelectNew` function. The parameter is passed as a Boolean `True`.

Rules `RowScrollSelectNew` does not arrange child rows under their related parent row keys. It is up to the `WHERE` clause in the SQL string to limit child rows to the parent record key. Select record should be defined and created using Application Designer and cannot be a Derived/Work record.

Returns Does not return a value

Example

```
RowScrollSelectNew(2, RECORD.MY_LOCATIONS,
RECORD.MY_LOCATION_EMP, RECORD.MY_LOC_EMPL_VW,
"WHERE SETID = :1 AND OPRCLASS = :2 AND LOCATION = :3",
MY_LOCATIONS.SETID, MY_LOCATIONS.OPRCLASS,
MY_LOCATIONS.LOCATION, True);
```

RowFlush

Description Used at the row level to remove a particular row of data from a scroll

Syntax

```
RowFlush(scrollpath, target_row)
```

ScrollPath is defined as:

```
[RECORD.level1_recname, level1_row,
[RECORD.level2_recname, level2_row]]
RECORD.target_recname
```

where

`RECORD.level1_recname` Represents the path to the target scroll area. When the target record is on scroll level 2, this parameter must precede the target_recordname. The leve1_recname requires the RECORD prefix.

`level1_row` Indicates the scroll level 1 row. The value is an integer and can be a variable or a constant. The parameter must be specified when the target record name is on scroll level 2.

`RECORD.level2_recname` Represents the path to the target scroll area. When the target record is on scroll level 3, the target_recordname must be preceded by a RECORD.level1_recname and RECORD.level2_recname.

`level2_row` This parameter represents data at scroll level 2. The value is an integer and can be a variable or a constant. The parameter must be specified when the target record name is on scroll level 3.

`target recordname` The target record name appears at the lowest scroll level. The target record name must be prefixed by RECORD. The target record may be on scroll level 1, 2, or 3. When the target record is on scroll level 2, the target record name must be prefixed with the RECORD.level1_recname, level1_row parameter. When the target is on scroll level 3, the target record name must be prefixed with the RECORD.level1_recname, level1_row, and the RECORD.level2_recname, level2_row.

`target_row` Identifies the row number to be removed from the specified target scroll area

Rules `RowFlush` does not remove rows from the database; it only removes them from the panel scroll buffer. To remove records from the panel scroll buffer as well as from the database, the `DeleteRow` function can be used because it performs both operations.

Returns Does not return a value

Example

```
If EMPLID <> PERSONAL_DATA.EMPLID Or
      EFFDT <> DIRECT_DEPOSIT.EFFDT Then
   RowFlush(RECORD.DIRECT_DEPOSIT, CurrentRowNumber(),
 RECORD.DIR_DEP_DISTRIB, CurrentRowNumber());
End-If;
```

UpdateValue

Description `UpdateValue` is commonly used in a scroll area to update the value of a field using a value parameter.

Syntax

```
UpdateValue (Scrollpath, target_row,
 [recordname.] fieldname, value )
```

ScrollPath is defined as:

```
[RECORD.level1_recname, level1_row,
[RECORD.level2_recname, level2_row]]
RECORD.target_recname
```

where

`RECORD.level1_recname` Represents the path to the target scroll area. When the target record is on scroll level 2, this parameter must precede the target_recordname. The leve1_recname requires the RECORD prefix.

`level1_row` Indicates the scroll level 1 row. The value is an integer and can be a variable or a constant. The parameter must be specified when the target record name is on scroll level 2.

`RECORD.level2_recname` Represents the path to the target scroll area. When the target record is on scroll level 3, the target_recordname must be preceded by a RECORD.level1_recname and RECORD.level2_recname.

`level2_row` This parameter represents data at scroll level 2. The value is an integer and can be a variable or a constant. The parameter must be specified when the target record name is on scroll level 3.

`target recordname` The target record name appears at the lowest scroll level. The target record name must be prefixed by `RECORD`. The target record may be on scroll level 1, 2, or 3. When the target record is on scroll level 2, the target record name must be prefixed with the RECORD.level1_recname, level1_row parameter. When the target is on scroll level 3, the target record name must be prefixed with the RECORD.level1_recname, level1_row, and the RECORD.level2_recname, level2_row.

`target_row` Identifies the row number in the specified target scroll area be to updated

`[recordname.] fieldname` The name of the field to be updated on the target row. Recordname is used when the function call is made from a record definition that is not the same as the recordname. Fieldname can reside on scroll level 1, 2, or 3.

`Value` Identifies the variable, constant, or record field that is moved to the corresponding target record

Rules The data type of the value parameter must be of a type compatible with the record field. The `UpdateValue` function updates the value of the field in the scroll. If the panel is canceled, no changes are written to the database.

Returns Does not return a value

Example

```
&NEW_DATE = %Date;
If EFF_STATUS <> "A" Then
   For &I = ActiveRowCount(RECORD.DIRECT_DEPOSIT)
       To 1 Step - 1;
         UpdateValue(RECORD.DIRECT_DEPOSIT, CurrentRowNumber(),
         RECORD.DIR_DEP_DISTRIB, &I,
         DIR_DEP_DISTRIB.LAST_UPDATE_DATE, &NEW_DATE);
   End-For;
End-If;
```

TotalRowCount

Description Produces the aggregate number of rows in a scroll area including deleted rows

Syntax

```
TotalRowCount (Scrollpath)
```

ScrollPath is defined as:

```
[RECORD.level1_recname, level1_row,
[RECORD.level2_recname, level2_row]]
RECORD.target_recname
```

where

`RECORD.level1_recname` Represents the path to the target scroll area. When the target record is on scroll level 2, this parameter must precede the target_recordname. The leve1_recname requires the RECORD prefix.

`level1_row` Indicates the scroll level 1 row. The value is an integer and can be a variable or a constant. The parameter must be specified when the target record name is on scroll level 2.

`RECORD.level2_recname` Represents the path to the target scroll area. When the target record is on scroll level 3, the target_recordname must be preceded by a RECORD.level1_recname and RECORD.level2_recname.

`level2_row` This parameter represents data at scroll level 2. The value is an integer and can be a variable or a constant. The parameter must be specified when the target record name is on scroll level 3.

`target recordname` The target record name appears at the lowest scroll level. The target record name must be prefixed by RECORD. The target record may be on scroll level 1, 2, or 3. When the target record is on scroll level 2, the target record name must be prefixed with the RECORD.level1_recname, level1_row parameter. When the target is on scroll level 3, the target record name must be prefixed with the RECORD.level1_recname, level1_row, and the RECORD.level2_recname, level2_row.

Rules TotalRowCount is similar to ActiveRowCount except that TotalRowCount includes deleted rows. Rows that are marked as deleted remain in the buffer until all system updates have been performed.

Returns A number that includes active as well as deleted rows.

Example

```
/* To obtain total number of rows at level 1 */
   &TOTAL_ROWS = TotalRowCount(RECORD.MY_LOCATIONS);
/* Total number of rows at level 2 */
   &TOTAL_ROWS_LEVEL2 = TotalRowCount(RECORD.MY_LOCATIONS,
   CurrentRowNumber(), RECORD.MY_LOCATION_EMP);
```

SQL functions

SQLExec

Description Executes an SQL command passed as a string from a PeopleCode program. The SQL string can contain bind variables, subselects, and joins. Data elements appearing in a `Select` statement are returned to the PeopleCode program as output and can be stored in variables or record fields.

Syntax

```
SQLExec (sqlcmd, bindvars, output)
```

`sqlcmd` Represents an SQL string passed by the PeopleCode program. It can contain references to both regular and inline bind variables.

`bindvars` Bind variables are the data elements referenced in the SQL string. There are two types of bind variables, regular and inline. When regular bind variables are used, each requires a corresponding variable name that replaces the :n reference in the SQL string. These variables appear outside the double quotes as

variable-1 [, variable-2, variable-3 …]

When inline bind variables are used, the variables are enclosed within the SQL string as

[:recordname1.]fieldname1 [, [recordname2.]fieldname2] ...

`output` Represents the column name (s) populated as a result of a `Select` statement. The output can be placed into variables or record fields. Each column selected requires a corresponding output variable or record field separated by commas. The two forms include

variable-1 [, variable-2, variable-3] …

or

[:recordname1.]fieldname1 [, [recordname2.]fieldname2] …

Rules `SQLExec` is one function where unpredictable results can occur if rules are not followed. Because `SQLExec` bypasses the Application Processor and heads directly to the database, no evaluation of the SQL string contained within quotes is performed. Record fields used as inline bind variables or output variables are evaluated by the Application Processor when they are not contained in the SQL string. When PeopleCode containing `SQLExec` statements are entered into the PeopleCode editor, any undefined record fields are represented by an error message during the syntax check or save operation. SQLExec statements containing inline bind variables are the exception. Because an inline bind variable is enclosed in quotes, an SQL statement which contains incorrect inline bind variables generates a runtime error message. A previously undefined output variable is created at runtime and does not generate an error.

A `SQLExec Select` statement retrieves one row of data only. When multiple rows are selected, only the first row is actually returned.

The maximum number of output variables when using `Select` is 64.

With `SQLExec`, `Updates`, `Inserts` and `Deletes` can be performed but can only be done in the following events:

```
SavePreChg
WorkFlow
SavePostChg
```

Application records referenced in a `SQLExec` statement require the PS_ prefix.

Returns Returns an optional Boolean. A `True` indicates the function ran successfully.

Example

```
/* Using UPDATE with a regular bind variable */
SQLExec("Update PS_MY_LOCATIONS SET EFFDT = %1",
MY_LOCATIONS.EFFDT);

/* Using an inline bind variable */
SQLExec("Update PS_MY_LOCATIONS SET EFFDT =
:MY_LOCATIONS.EFFDT");
```

String functions

Lower

Description Converts the uppercase characters of the field or variable to lower case and returns them as a `String` data type

Syntax

```
Lower (string)
```

Rules Numeric, punctuation and other non-letter values are not changed

Returns A lowercase string

Example

```
&MY_STRING = "THIS STRING BECOMES LOWER CASE";
&NEW_STRING = Lower(&MY_STRING);
```

LTrim

Description Function is used to remove any leading characters identified in string2 from string1

Syntax

```
Ltrim (string1 [, string2])
```

Rules When string2 is not supplied, all leading blanks from string1 are removed. When string2 is supplied, the function is terminated when characters found in string1 do not match those found in string2.

Returns A string with leftmost characters in string2 removed or blanks when string2 is not supplied.

Example

```
&STREET_ADDRESS = ",##@&100 Main Street";
&STREET_ADDRESS = LTrim(&MY_STRING, ",.#@&");
/* &STREET_ADDRESS Now contains 100 Main Street */
```

RTrim

Description Function is used to remove any rightmost characters identified by trim_str from the source string

Syntax

```
RTrim (source_str, [, trim_str] )
```

Rules Works from right to left removing trailing characters defined in trim_str. When trim_str is not supplied, any rightmost blanks are removed.

Returns A string with leftmost characters in string2 removed or blanks when string2 is not supplied.

Example

```
&DEPARTMENT_DESCR = "Software development & Web
Services,,,,,,";
&DEPARTMENT_DESCR = RTrim(&DEPARTMENT_DESCR, ",");
```

Upper

Description Converts the characters appearing in a text string to upper case values

Syntax

```
Upper (string)
```

Rules Characters such as numeric, punctuation, and other non-letter values are not changed

Returns Returns a string containing uppercase values.

Example

```
&LAST_NAME = "picard";
&LAST_NAME_SRCH = Upper(&LAST_NAME);
/* Value of &LAST_NAME_SRCH = PICARD */
```

Trace Control functions

SetTracePC

Description Controls PeopleCode `Trace` based on parameter values passed.

Syntax

```
SetTracePC (n)
```

Rules Takes one parameter, which represents the trace settings used in producing the output trace file. When multiple trace options are required, each option number is added, and the sum is passed to the function. The options available to `SetTracePC` are shown in table E.2.

By default `SetTracePC` produces a file named DBG1.TMP in the Windows Temp directory. A unique file name can be specified if necessary, and this can be done from within the configuration manager trace option.

Returns Does not return a value

Table E.2 SetTracePC options

Option #	Description
1	This option traces the program that is executed. It includes options 64, 128 and 256 specified below.
2	Lists the entire program.
4	Displays the outcomes of assignments made to variables.
8	Identifies the values retrieved for all variables.
16	Identifies the contents used in the internal stack.
64	This trace option identifies when each program is started.
128	Identifies the calls made to external PeopleCode routines.
256	Identifies the calls made to internal PeopleCode routines.
512	Displays the value of parameters passed to a function.
1024	This option displays the values of parameters at the conclusion of a function call.

Transfer functions

SetNextPanel

Description `SetNextPanel` identifies a panel name that will be transferred control to when the operator activates the F6 or presses the NextPanel toolbar icon.

Syntax

```
SetNextPanel (panelname)
```

Rules Verifies that the panel identified by panelname is available on the current active menu

Returns Returns an optional Boolean value based on the success or failure of the function call

Example

```
If &RETURN_CODE = 0 Then
   SetNextPanel("MY_APPLCTN_TBL");
Else
   SetNextPanel("MY_USER_TBL");
End-If;
```

TransferPanel

Description Transfers control to the next panel in the panel group, the panel name supplied to the function, or to the panel identified by a previous `SetNextPanel` function.

Syntax

```
TransferPanel ( [panel_name] )
```

or

```
TransferPanel ( )
```

Rules The panel transferred to must exist in the current panel group. When the function is called from events outside of save processing (`SavePreChg`, `SavePostChg`), any PeopleCode statements following the `TransferPanel` function are not executed and processing is halted.

Returns Returns an optional Boolean value based on the success or failure of the function call.

Example

```
If &RETURN_CODE = 0 Then
   SetNextPanel("MY_APPLCTN_TBL");
Else
   SetNextPanel("MY_USER_TBL");
End-If;
TransferPanel();

/* Can also be written */

If &RETURN_CODE = 0 Then
   &NEXTPANEL = "MY_APPLCTN_TBL";
Else
   &NEXTPANEL = "MY_USER_TBL";
End-If;
TransferPanel(&NEXTPANEL);
```

META-SQL FUNCTIONS

Meta-SQL functions are used in SQL strings. They expand in these strings to become platform-specific parameters in the SQL statements. SQL strings are used in the `SQLExec` as well as scroll functions that accept an SQL string. Meta-SQL can also be implemented when constructing dynamic views or Application Engine statements.

Table E.3 Selected Meta-SQL functions

Function	Description
`%CurrentDateIn`	This is an `In` function that becomes a platform-specific SQL string. The string can be used to represent current date in a `Select`, `Update`, or `Insert` statement.
`%CurrentDateOut`	An `Out` function that can be used as the current date in the `Select` clause of an SQL string
`%CurrentDateTimeIn`	An `In` function that becomes a platform-specific SQL string. The string can be used as a `DateTime` value in a `Select`, `Update`, or `Insert` statement.
`%CurrentDateTimeOut`	`%CurrentDateTimeOut` is an out function that can be used as the current `DateTime` value in the `Select` clause of an SQL string.
`%CurrentTimeIn`	This is an `In` function that becomes a platform-specific SQL string. The string is used as current time in a `Select`, `Update`, or `Insert` statement.
`%CurrentTimeOut`	An `Out` function that can be used as the current time in the Select clause of an SQL string
`%DateAdd`	Returns a date after adding the add_days parameter to date_from. syntax: `%DateAdd` (date_from, add_days) add_days is an integer that can have a negative value and is added to date_from.

`%DateDiff`	Identifies the difference between two dates
	syntax: `%DateDiff` (date_from, date_to)
	The difference between date_from and date_to is returned as an integer value. When a date literal is used, it must be passed as a `%DateIn`.
	Example
	`&Difference = %DateDiff(INCIDENT_DT, CLOSE_DT);`
	`&Difference = %DateDiff (CLOSE_DT, %DateIn('1999-05-31'));`

`%DateIn`	An `In` function that becomes a platform-specific SQL string. The function accepts a date value parameter or a date literal in the format YYYY-MM-DD. `%DateIn` is used in SQL statements such as `Select`, `Insert`, and `Update` that require a date bind variable or date literal. syntax %DateIn(date)
`%DateOut`	An `Out` function that can be used as the date in the `Select` clause of an SQL string syntax: %DateOut(date)
`%TimeIn`	This is an `In` function that becomes a platform-specific SQL string. The string is used as the time value in a `Select`, `Update`, or `Insert` statement. The time parameter passed can be a time variable or a literal in the form hh:mm:ss.ssssss [{AM|PM}].

Table E.3 Selected Meta-SQL functions (continued)

Function	Description
`%TimeOut`	An `Out` function that can be used as the time in the `Select` clause of an SQL string
`%Substring`	This function references only the portion of the string identified by source_str. The starting position is identified by start and is relative to 1. Length represents the number of characters to be referenced. `%Substring` can be used to extract or compare a selected area of a string. syntax %Substring (source_str, start, length)
`%TrimSubstr`	This function is similar to `%Substring` and can be used to extract or compare a selected area of a string. The difference is that any trailing blanks in the string referenced by source are removed from the target substring. syntax %TrimSubstr (source_str, start, length)

APPENDIX F

Application Engine functions

Eight basic functions or macros can be utilized in Application Engine statements: &BIND, &CLAUSE, &CLEARCURSOR, &EXECUTE, &MSG, &&RECORD, &ROUND, and &SELECT. We've already covered the most common functions in our exercises (&BIND, &MSG, and &SELECT). The only macro is &&RECORD.

&BIND

Purpose Retrieves an individual field value from the cache record.

Syntax

```
&BIND(cache_field [,NOQUOTES] [,NOWRAP] [,STATIC])
```

Rules The &BIND function can be used almost anywhere in an SQL statement. It cannot be used in a Select statement Result Set field list.

A character field is returned enclosed in quotes unless the optional NOQUOTES parameter is used.

Date fields are automatically enclosed (or "wrapped") within the %DATEIN or %DATEOUT Meta-SQL functions unless the optional NOWRAP parameter is specified.

When the STATIC parameter is specified, Application Engine resolves the &BIND variable before compiling the SQL statement. This is useful when creating dynamic SQL statements.

Example

```
&SELECT(COUNTER)
SELECT COUNT(*)
  FROM PS_&BIND(RECNAME,NOQUOTES,STATIC)
```

The example is the same used in exercise #3 of our tutorial. When using a RECNAME of JOB, the following SQL statement is compiled:

```
SELECT COUNT(*) FROM PS_JOB
```

The value of JOB is not enclosed in quotes due to the NOQUOTES parameter and can therefore be concatenated properly with the PS_ prefix. The STATIC parameter tells Application Engine to resolve the &BIND variable before compiling the statement.

The &SELECT portion of this statement is described in the &SELECT section.

&CLAUSE

Purpose Similar to a COBOL copybook (or #Include in SQR). When used in a statement it is replaced with the contents of the Application Engine statement specified in the &CLAUSE function. One of the main uses of the &CLAUSE function is retrieving predefined column lists and substituting them in the calling statement. This is useful for Select lists or Insert statements. There are several parameters that can be used with &CLAUSE to increase its' flexibility.

Syntax

```
&CLAUSE(product, application, section, step, type [,parm1] …
 [,parm9])
```

Rules &CLAUSE must point to a valid statement designated by the fully qualified Application Engine statement name of product, application, section, step, and type. No validation is performed. Any errors will be recognized at run-time.

The actual clause section that's retrieved may have optional &P(n) parameter variables embedded within it. A parameter value must be passed to it with the &CLAUSE function. Up to nine parameters may be passed. Several symbolic parameters may be used in an &CLAUSE function:

&COMMA Since a physical comma (or ',') would not be interpreted as an actual parameter, the &COMMA symbolic can be used.

&SPACE This symbolic represents a blank or space. If the retrieved clause section uses &P(n) but isn't required a space can be passed to resolve them using &SPACE.

&RPAREN A right parenthesis may be passed to the specified "clause section." Once again, the symbolic must be used. A physical right parenthesis would not be interpreted as a parameter that needs to be passed.

One of the primary uses of parameters in a &CLAUSE function is to pass a synonym to be used as a column prefix.. The table synonym must be passed with a

period like 'A.' or 'B.' —without the quotes. The &CLAUSE function would then return the columns with the desired prefixes to your current statement.

Example Assuming we defined a "clause section" for product = AE, Appl ID = SAMPLE, section = COMMON, step = CITYINFO, and statement type = S. The column list is entered in the statement text box as follows:

```
&p(1)CITY
,&p(1)STATE
,&p(1)ZIP
```

The &CLAUSE function allows us to substitute the above column list anywhere in our program. Using parameters, we can tailor the column list to our particular needs with prefixes. You'll notice the &P(n) prefix variable contains a '1' for all three columns. This means the first parameter passed in the &CLAUSE function is used as the prefix for all three columns. Using &p(n), the n represents the nth parameter passed in the &CLAUSE.

```
INSERT INTO ps_user_cityinfo
( &CLAUSE(AE, SAMPLE, COMMON, CITYINFO, S, &SPACE) )
SELECT &CLAUSE(AE, SAMPLE, COMMON, CITYINFO, S, A.)
  FROM ps_temp_cityinfo A
```

The &CLAUSE is used twice: Once for the Insert Column list, which doesn't allow prefixes, and once for the Select Column list, which in this case uses a prefix of 'A'. When the SQL statement is resolved, it appears as:

```
INSERT INTO ps_user_cityinfo
(CITY, STATE, ZIP)
SELECT A.CITY, A.STATE, A.ZIP
  FROM ps_temp_cityinfo A
```

Also, see the &&RECORD macro for similar functionality.

&CLEARCURSOR

Purpose A re-used statement may need to be recompiled during execution of the program. The &CLEARCURSOR function accomplishes this and resets any &BIND variables in the statement that use the STATIC option.

Syntax

```
&CLEARCURSOR([product, ] [application, ] section, step,
 type)
```

Rules This function must be located at the start of the statement. There may be no other functions or commands in the statement.

Example

```
&CLEARCURSOR(BI, BIIVC000, DUEDATE, SETDATE, D)
```

This recompiles the `DO Select` statement in the SETDATE step. The step is found in the DUEDATE section of the billing application BIIVC000.

NOTE Refer to the section describing the Statement Definition panel in appendix F for an explanation of re-used statements.

&EXECUTE

Purpose Database-specific commands may be executed with this function. Generally, this means any SQL statement that cannot be executed directly using the `Update/Insert/Delete` SQL statement type.

Syntax

```
&EXECUTE( [/] )
   command_1 {; | /} …
   command_n {; | /}
```

Rules The `Update/Insert/Delete` statement type must be used. This function must be located at the start of the statement. No other functions or commands may exist in the statement.

Application Engine expects each command within the `&EXECUTE` function to be delimited with a semi-colon. The optional forward slash (/) parameter overrides this convention and allows the use of a procedural language such as Oracle's PL/SQL to be used. Since the commands within a PL/SQL block are normally terminated by a semi-colon, the forward slash override avoids any conflict. The forward slash would then be required at the end of the `&EXECUTE` statement.

Example

```
&EXECUTE(/)
declare
   ctr   integer:= 0;
begin
   while ctr = 0 loop
      ctr   = ctr + 1;
      update ps_installation_ar   set st_id_num = ctr;
   end loop;
end;
/
```

The forward slash (/) tells Application Engine to execute the entire PL/SQL block. No conflicts result due to the semi-colon.

&MSG

Purpose The &MSG function writes a message to the message log.

Syntax

```
&MSG( [Message_Set_Number], Message_Number, [Parm_1],….
 [Parm_n] )
```

Rules The &MSG function always uses an SQL statement type of Update and must be the first and only function or command in the statement.

Example

```
&MSG(,1,'Hello World')
```

The example is the same as that used in exercise #1. Since the Message Set Number is excluded, it defaults to the Message Set Number specified on the Application Engine definition panel. Message Number 1 is passed a string value of "Hello World". This string value is used in place of the %1 substitution variable defined in the message catalog entry.

&&RECORD

Purpose The &&RECORD macro inserts all the field names of the specified record into your statement. The optional parameter can be used to assign a column synonym when the entire record is required in your statement(s). This is a quick alternative to the &CLAUSE function.

Syntax

```
&&RECORD(record [, parm_1] )
```

Rules You must use a valid RECNAME.

Example

```
INSERT INTO ps_customer_tao
SELECT &&RECORD(CUSTOMER)
  FROM ps_customer
```

Using &&RECORD, the Select statement uses all the columns in the Customer record as they exist in Application Designer. This example assumes the CUSTOMER_TAO record matches the Customer record exactly.

&ROUND

Purpose When Multi-Currency is activated, this function can be used to round numeric fields to the currency precision specified under Define General Options.

Syntax

```
&ROUND(field)
```

Rules The Multi-Currency option must be specified.

To set the Multi-Currency option

Go →PeopleTools →Utilities →Use →PeopleTools Options

To set the currency precision:

Go →Define Business Rules →Define General Options →Use A-D → Currency Code

Example

```
UPDATE ps_user_tmp
   SET USER_AMT1 = &ROUND(USER_AMT1)
```

This example updates the table with the USER_AMT1 value rounded to the appropriate currency precision.

&SELECT

Purpose Updates the cache field with the value assigned by the corresponding SQL `Select` statement

Syntax

```
&SELECT(cache_field_1 [,cache_field_2] [,cache_field_x] )
SELECT field_1 [,field_2] [,field_x]
```

Rules `&SELECT` is used in tandem with an SQL `Select` statement immediately following.

- The number of cache fields must match the number of fields in the SQL `Select`.
- The datatypes of corresponding cache and SELECT fields must match.
- If NO rows are returned by the SQL `Select` statement, the cache fields are assigned a value of `zero` or `blank`, depending on the datatype.

Example

```
&SELECT(COUNTER)
SELECT COUNT(*)
  FROM PS_PERSONAL_DATA
```

The example is the same as that used in exercise #2. The record count is selected from the PERSONAL_DATA table (lines 2 and 3). The `&SELECT` function (line 1) assigns the record count to the cache field COUNTER.

index

System variables

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

ERP/PEOPLESOFT

Essential Guide to PeopleSoft Development and Customization

Tony DeLia ♦ Galina Landres ♦ Isidor Rivera ♦ Prakash Sankaran

PeopleSoft is a collection of software modules for common business functions. It is also a toolset for companies to modify, customize, and enhance the modules to suit their needs. Essential Guide to PeopleSoft Development and Customization is a comprehensive reference and tutorial that covers PeopleSoft 7.5 and gives an overview of new features in release 8.0.

Both novice and experienced programmers will benefit from this book's coverage of all aspects and elements of PeopleSoft application development. Topics run from the basics of the PeopleSoft development paradigm to the secrets of such powerful tools as SQR, Application Designer, PeopleCode, PeopleSoft COBOL, and the ins and outs of PeopleSoft customization. One of the most powerful but least understood PeopleSoft tools, the Application Engine, gets special coverage, including dynamic SQL, decision logic, and dynamic sections.

This book's broad range and detailed, practical advice will give you the skills necessary to compete in the PeopleSoft marketplace for years to come.

What's inside

- Basics of application development
- PeopleCode
- PeopleTools
- Structured Query Report Writer (SQR)
- PeopleSoft for on-line applications
- PeopleSoft customization with real examples
- The Application Engine and Application Designer
- PeopleSoft COBOL and the PTPSQLRT module

Together, Tony DeLia, Galina Landres, Isidor Rivera and Prakash Sankaran, have a very large amount of experience implementing PeopleSoft applications. This book is the end result of knowledge they accumulated in the past ten years since the first release of PeopleSoft.

"There's a vital need for this book to help answer our questions and aid us on the job."

—Steven Britt
Tropical Shipping, LTD

"... it deals with development pitfalls head-on so you know how to deal with them—or avoid them altogether."

—Peter Choi
PeopleSoft/Oracle DBA
APG Solutions & Technologies

"I will definitely recommend the book for those new to PeopleSoft. It is also a great reference for experienced developers."

—Cary Cloud
Technology Solutions Company

www.manning.com/delia

MANNING $64.95 US/$94.95 Canada

9 781884 777929 56495
ISBN 1-884777-92-9