Fueling the Workflow Engine

Dan Stober

Intermountain Healthcare

Objectives:

1. Get to know the basic tables used to store Workflow setup and runtime data. Learn how the data are stored, how to join the tables, and how to query the tables..
2. Learn how to use Oracle's Workflow scripts and APIs to launch and interact with your processes
3. Learn how to use Workflow to call your own custom functions and procedures
4. Understand the different types of code that you can write

OUTLINE

UNIT / CHAPTER / SECTION

Unit I - Understanding the Workflow tables 2
Chapter 1 – The roadmap: the definition tables 3
Chapter 2 – The traffic: the runtime tables 15
Unit II - Oracle's delivered Scripts and APIs 19
Chapter 3 – Workflow Scripts 19
Chapter 4 – APIs 23
Unit III – Writing PLSQL to Implement Function Activities 30
Chapter 5 – The call signature 32
Chapter 6 – Error Handling 38
Chapter 7 – Three examples 42
Unit IV – Other PLSQL Code for Workflow 52
Chapter 8 – A concurrent program 52
Chapter 9 – Selector Functions 54
Chapter 10 – Document Type Attributes 54

INTRODUCTION

PURPOSE AND PREREQUISITES

"Fueling the Workflow Engine: How to Write PLSQL For Oracle Workflow" will help you build on your existing knowledge of Oracle Workflow. This paper is focused on how to write PLSQL for Workflow. We'll explore procedures that implement Workflow function activities, procedures to populate document type attributes, and procedures that create selector functions, and provide annotated examples of each. Along the way, we'll examine the workflow data model, understand some of the available APIs, and see a few of the scripts provided by Oracle. I'll also point out a few "gotchas" along the way and will provide some avenues for you to explore if you want to learn more

 © Copyright 2008 by Daniel Stober

To get the most out of this paper, you should have some knowledge of and exposure to Oracle Workflow. Perhaps you've used the Workflow Builder tool to create or modify a process definition, or maybe you've used the Workflow Administration pages to support Workflows in your organization. Additionally, knowledge of PLSQL is very helpful to understanding the topics we'll cover, as it will serve you well in fully grasping the concepts and examples in this paper.

UNIT I: UNDERSTANDING THE WORKFLOW TABLES

Odds are, your first exposure to Oracle Workflow came in an encounter using the Workflow Builder or working in the administration screens. While these tools provide a pleasant, user-friendly experience, they also make it very easy to overlook an important fact: underneath the fancy user interface, Oracle Workflow is a database application. It is an application that runs on a database, and the entire Oracle Workflow application utilizes database tables as the basis of its operation. The process definitions created in the Workflow Builder are stored in tables. The attribute values are stored in tables. The status information about the various activities as a process runs is stored in tables. Even the stored procedures are stored in the database! But then, would you expect anything less from a product created by the largest database company on Earth?

With that said, it should be apparent that if you really want to get to know Workflow and discover how it works, you have to understand its table structures. When you know what the data in the tables look like, you can write queries and troubleshoot problems. In the following paragraphs, we're going to examine a few of the key tables that comprise the Oracle Workflow application. I'll try to help you make sense of the table names that, at first glance, may appear to be quite similar, and I'll show what pieces of information are stored where.

When delving into the Workflow tables, a good dichotomy to use as a starting point is to group the tables into two major categories: the tables that hold setup information and the tables that store information about processes in progress. For the purposes of this discussion, we'll call the tables with the setup data "definition tables," and the others will be the "runtime tables." As a point of reference, when you use the Workflow Builder to create or modify a workflow process, you are writing records to the definition tables, but when a process instance is launched and is wending along through the various activities, its progress is tracked in the runtime tables.

Definition Tables ***Used for set up***	**Runtime Tables** ***Information about running processes***
• wf_item_types • wf_item_attributes • wf_activities • wf_activity_attributes • wf_activity_attr_values • wf_messages • wf_message_attributes • wf_process_activities • wf_activity_transitions	• wf_items • wf_item_attribute_values • wf_item_activity_statuses • wf_item_activity_statuses_h • wf_notifications

Figure 1

A sampling of some of the important database tables behind Oracle Workflow. This is not a complete list.

Because information about roles and users within the E-Business Suite comes from fnd_users and responsiblities, that information will not be included in this discussion.

© Copyright 2008 by Daniel Stober

Contents:

THE DEFINTION TABLES

WF_ITEM_TYPES	6
WF_ITEM_ATTRIBUTES	7
WF_ ACTIVITIES	7
WF_ACTIVITY_ATTRIBUTES	9
WF_ACTIVITY_ATTR_VALUES	10
WF_MESSAGES	10
WF_MESSAGE_ATTRIBUTES	11
WF_PROCESS_ACTIVITIES	11
WF_ACTIVITY_TRANSITIONS	13
WF_LOOKUP_TYPES	14
WF_LOOKUPS_TL	15

THE RUNTIME TABLES

WF_ITEMS	15
WF_ITEM_ACTIVITY_STATUSES	16
WF_ITEM_ACTIVITY_STATUSES_H	17
WF_NOTIFICATIONS	18
WF_ITEM_ATTRIBUTE_VALUE	18

Chapter 1: THE DEFINITION TABLES

Our look at the database tables will begin with the definition tables. Just as a roadmap describes what a route looks like and what alternatives might be available, these are the tables that describe what an item_type looks like. They answer questions like, "What attributes can it have?", "What datatypes do those attributes contain?", and "What processes are available, and what activities make up those processes?" Think of the definition tables as the roadmap that the Workflow engine will use to route a process instance from start to finish.

Within most of the definition tables, the field called "name" represents the internal name of the object being stored. For example, in wf_item_types, which is the table used to store item_types, the internal name is the value stored in the "name" field. In wf_item_attributes, the "name" field contains the internal name of the attribute. Generally, that field will comprise all or part of the primary key of its table, which allows the database to enforce uniqueness upon the values entered.

LOCALIZATIONS SCHEME

When you call your credit card company to check your balance, the first voice you hear probably says something like, "To hear this message in English, press 1. *Para oir esta mensaje en español, marquee el 2*." After you have made your selection, the system probably will play its main menu message. If you chose English, you'll hear that menu in English, but if you selected Spanish, the system will play the main menu in that language.

 © Copyright 2008 by Daniel Stober

If you take a close look at a complete listing of the definition tables, you may notice that there are many groups of tables that seem to have similar names and seem to share many of the same columns in common. For example, the first table that we are going to cover in this section will be wf_item_types. But, there is another table in the data dictionary called wf_item_types_tl that seems to store a lot of the same information. Alongside these two is a view called wf_item_types_vl.

So what's going on here?

This naming scheme is part of Workflow's implementation of user specific languages and localization settings. Remember, Oracle is a company that does business around the world, and among its customers are many multi-national corporations that have offices and employees around the globe. Workflow's localization schemes allow developers to create a single process that can be deployed in many locations, and supply messages and display names for the users in the local language. Just like the credit card company can serve up multiple version of the same message, Oracle Workflow can, too.

To expand on this theme, we'll examine wf_messages, wf_messages_tl, and wf_messages_vl, all of which are used to store messages that will become part of a notification. (The function of these tables is covered a little bit later in this section on page 10.)

The base table, wf_messages, contains fields whose values remain the same regardless of the locality. These are the fields that are needed to support joins and for Workflow's internal administration. Fields like "type" (message_type), "name", "customization_level", and "protection_level" are fields that all message records will share in common, regardless of the localization.

In Workflow's localizations scheme, the table names ending with "_tl" are the tables providing the translations for localizations. They contain the user-friendly field values for display names in the local language. In the case of wf_messages_tl, you'll find the fields which actually contain the messages in this table.

.

For example, let's say that you designed a Workflow process for a company which does business in the United States and Canada. Your process might include a message named SHIP_CONFIRM that is intended to be used in notifications informing a user that his or her order has been shipped,. But, since you have customers in Québec, you need to have versions of your message in two languages. You'll have to create a French language shipping confirmation message to serve your Francophone customers.

```
SQL> select userenv('LANG') from dual
  2  /

USERENV('LANG')
---------------
US

1 row selected.

SQL>
```

Figure 2

Finding the language preference for the user in the current session.

To continue this example, once you have finished creating your two messages, one record would be inserted into wf_messages, but you would end up with two records in the table wf_messages_tl, and both of them would be named SHIP_CONFIRM. The difference between the two records is that one would have a language code value of "US" and the other would contain "FRC", for French Canadian. Although both of those records would have the same name, each would have a completely different message, composed in the appropriate language.

The final piece in this puzzle that makes the whole scheme work, is the localization view, identified by a name that ends with the string "_vl" . This view joins the multiple language records from the _tl table with the session variable which contains the user's own language preference. The result of this join is that a single record is returned – the record from the wf_messages_tl table with the language that matches the user's preference setting.

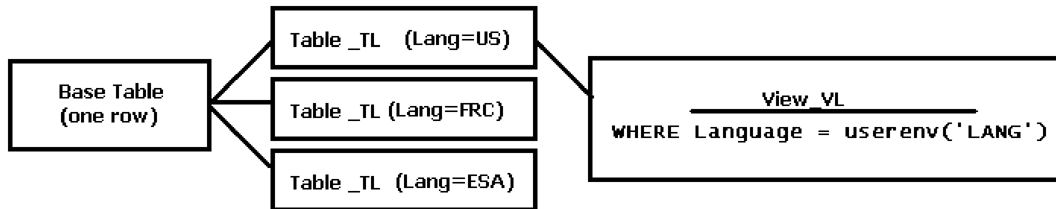

Figure 3

Graphic representation of the table relationships in the localization scheme. A single row in the base table yields as many rows in the _TL as the number of languages in the installation. Those records are matched with session information to resolve to a single record in the language of the user's session to create the view _VL.

The credit card company answers your call by asking which language you prefer, and then plays versions of its messages that match that preference. Similarly, Oracle Workflow gets the language preference of the user from the session variables, and then sends messages that match that preference by using the "_vl" view.

Most of the table definitions we'll see in this section follow the same pattern and operate in precisely the same way. However, to avoid redundancy, the way that I'll handle them as we encounter them in the discussion will be to list all three, the name of the base table, the name of the translation table, and the name of the localization view, above the paragraph, but then to treat the three objects as one in the explanatory text. The presence of all three names will alert the reader as to the existence of the other tables and views, even as the discussion treats the functionality as a single entity.

Along with each grouping, I have included describes for each of the tables to the right of each explanation. Those descriptions contain the field listing for each of the two tables, but not for the view. Generally, you can assume that any field which appears in either of the two table definitions also appears in the view.

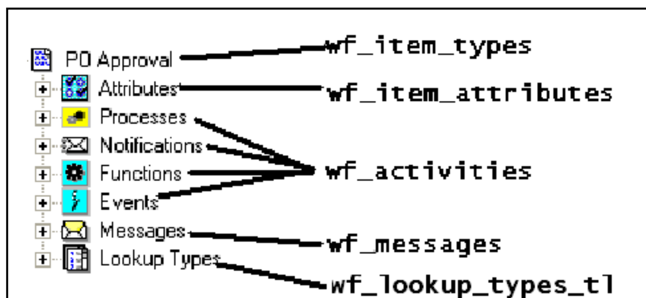

Figure 4

Mapping the objects in the Workflow Builders to the set-up tables that store them

Much like the objects in the Workflow Builder, the definition tables can be visualized in a hierarchical association. We'll start at the top.

 © Copyright 2008 by Daniel Stober

WF_ITEM_TYPES

WF_ITEM_TYPES _TL

WF_ITEM_TYPES _VL

```
SQL> desc wf_item_types
 Name                            Null?    Type
 ------------------------------- -------- ----
 NAME                            NOT NULL VARCHAR2(8)
 PROTECT_LEVEL                   NOT NULL FLOAT(*)
 CUSTOM_LEVEL                    NOT NULL FLOAT(*)
 WF_SELECTOR                              VARCHAR2(240)
 READ_ROLE                                VARCHAR2(320)
 WRITE_ROLE                               VARCHAR2(320)
 EXECUTE_ROLE                             VARCHAR2(320)
 PERSISTENCE_TYPE                NOT NULL VARCHAR2(8)
 PERSISTENCE_DAYS                         FLOAT(*)
 SECURITY_GROUP_ID                        VARCHAR2(32)

SQL> desc wf_item_types_tl
 Name                            Null?    Type
 ------------------------------- -------- ----
 NAME                            NOT NULL VARCHAR2(8)
 LANGUAGE                        NOT NULL VARCHAR2(30)
 DISPLAY_NAME                    NOT NULL VARCHAR2(80)
 PROTECT_LEVEL                   NOT NULL FLOAT(*)
 CUSTOM_LEVEL                    NOT NULL FLOAT(*)
 DESCRIPTION                              VARCHAR2(240)
 SOURCE_LANG                     NOT NULL VARCHAR2(4)
 SECURITY_GROUP_ID                        VARCHAR2(32)
```

At the top of the hierarchy of all the definition tables is wf_item_types.

The wf_item_types table contains one record for each item_type created. The eight character name of the item_type itself, the value that you may know as the "Internal Name" of the item, functions as the primary key for this table. All other tables are joined using the VARCHAR2 item_type via a foreign key relationship. By itself, this table is pretty basic, containing just a minimal few fields necessary to establish an item_type. The "meat" of an actual item results from joining records in the other definition tables.

The name of the PLSQL procedure which implements selector function is stored in the "wf_selector" field. We will cover selector functions – how they work and how to create one – in greater detail later on the paper.

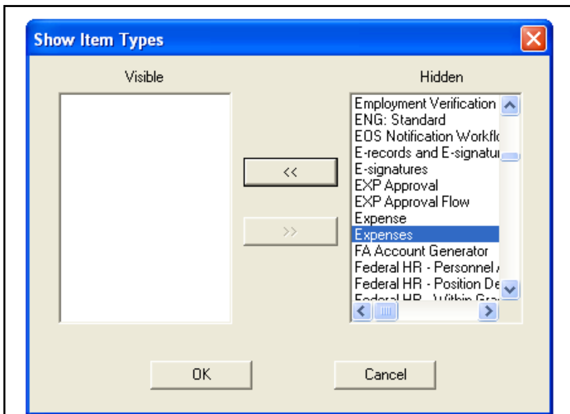

Figure 5

Trying to find the right item type in the Workflow Builder can be a challenge – especially if you don't know the display name of the item. Is it "Expense" or "Expenses"? Find out by querying wf_item_types_tl.

Do you remember the first time you even opened up the Workflow Builder and looked for an item type definition in the database? You saw a daunting list of item types, and it was probably very difficult to find the one you wanted. Perhaps you were looking for APEXP, but couldn't find it in the list. Of course, you probably now know that the reason you had trouble locating it is because the Workflow Builder exhibits item types using the user-friendly display name, not the internal name, so to find APEXP you had to look for "Expenses".

Of course, "user-friendly" is a subjective term – it's only friendly if you know what you're looking for! The nexus between the internal name and the display name of the item type is formed here. Notice that the display name is a field in the "_tl" table, but not the base table. If you know the internal name for the item type and need to find the display name, look here.

You may have noticed the names of three different objects are listed above this paragraph. This is just the first example of how the scheme of two tables and single view that Workflow uses to implement localization will be represented in this paper. In order to highlight the distinction, the name of the view appears in italics. Read the section above if you are unclear.

 © Copyright 2008 by Daniel Stober

WF_ITEM_ATTRIBUTES

WF_ITEM_ATTRIBUTES_TL

WF_ITEM_ATTRIBUTES_VL

```
SQL> desc wf_item_attributes
 Name                            Null?    Type
 ------------------------------- -------- ----
 ITEM_TYPE                       NOT NULL VARCHAR2(8)
 NAME                            NOT NULL VARCHAR2(30)
 SEQUENCE                        NOT NULL NUMBER
 TYPE                            NOT NULL VARCHAR2(8)
 PROTECT_LEVEL                   NOT NULL NUMBER
 CUSTOM_LEVEL                    NOT NULL NUMBER
 SUBTYPE                                  VARCHAR2(8)
 FORMAT                                   VARCHAR2(240)
 TEXT_DEFAULT                             VARCHAR2(4000)
 NUMBER_DEFAULT                           NUMBER
 DATE_DEFAULT                             DATE
 SECURITY_GROUP_ID                        VARCHAR2(32)

SQL> desc wf_item_attributes_tl
 Name                            Null?    Type
 ------------------------------- -------- ----
 ITEM_TYPE                       NOT NULL VARCHAR2(8)
 NAME                            NOT NULL VARCHAR2(30)
 LANGUAGE                        NOT NULL VARCHAR2(30)
 DISPLAY_NAME                    NOT NULL VARCHAR2(80)
 PROTECT_LEVEL                   NOT NULL NUMBER
 CUSTOM_LEVEL                    NOT NULL NUMBER
 DESCRIPTION                              VARCHAR2(240)
 SOURCE_LANG                     NOT NULL VARCHAR2(4)
 SECURITY_GROUP_ID                        VARCHAR2(32)
```

In this table, the Workflow designer establishes the attributes that will be available to the item_type. The entries in this table correspond to the “Attributes” subheading in the Workflow Builder.

An item attribute is akin to a variable in a Workflow definition. An item attribute can hold values that are specific to the process instance or which may change at run time.

Each item attribute is assigned a datatype, such as “Character”, “Number”, or “Date”. That value is stored in the “type” field.

There are three fields to hold a default value, but only one of them will be populated for any item attribute, depending upon the datatype. For example, if you create an item attribute with a datatype of “Number”, and then supply a default value, that value would be stored in the “number_default” field.

The “format” field stores information about a format mask that should be applied to number or date values, and the “subtype” field contains “SEND” or “RECEIVE”.

WF_ ACTIVITIES

WF_ ACTIVITIES_TL

WF_ACTIVITIES_VL

```
SQL> desc wf_activities
Name                            Null?    Type
------------------------------- -------- ----
ITEM_TYPE                       NOT NULL VARCHAR2(8)
NAME                            NOT NULL VARCHAR2(30)
VERSION                         NOT NULL NUMBER
TYPE                            NOT NULL VARCHAR2(8)
RERUN                           NOT NULL VARCHAR2(8)
EXPAND_ROLE                     NOT NULL VARCHAR2(1)
PROTECT_LEVEL                   NOT NULL NUMBER
CUSTOM_LEVEL                    NOT NULL NUMBER
BEGIN_DATE                      NOT NULL DATE
END_DATE                                 DATE
FUNCTION                                 VARCHAR2(240)
RESULT_TYPE                              VARCHAR2(30)
COST                                     NUMBER
READ_ROLE                                VARCHAR2(320)
WRITE_ROLE                               VARCHAR2(320)
EXECUTE_ROLE                             VARCHAR2(320)
ICON_NAME                                VARCHAR2(30)
MESSAGE                                  VARCHAR2(30)
ERROR_PROCESS                            VARCHAR2(30)
ERROR_ITEM_TYPE                 NOT NULL VARCHAR2(8)
RUNNABLE_FLAG                   NOT NULL VARCHAR2(1)
FUNCTION_TYPE                            VARCHAR2(30)
EVENT_NAME                               VARCHAR2(240)
DIRECTION                                VARCHAR2(30)
SECURITY_GROUP_ID                        VARCHAR2(32)

SQL> desc wf_activities_tl
Name                            Null?    Type
------------------------------- -------- ----
ITEM_TYPE                       NOT NULL VARCHAR2(8)
NAME                            NOT NULL VARCHAR2(30)
VERSION                         NOT NULL NUMBER
DISPLAY_NAME                    NOT NULL VARCHAR2(80)
LANGUAGE                        NOT NULL VARCHAR2(30)
PROTECT_LEVEL                   NOT NULL NUMBER
CUSTOM_LEVEL                    NOT NULL NUMBER
DESCRIPTION                              VARCHAR2(240)
SOURCE_LANG                     NOT NULL VARCHAR2(4)
SECURITY_GROUP_ID                        VARCHAR2(32)
```

Wf_activities is the Workflow definition table for *all* activities comprising an item_type. Functions, notifications, processes, and events all are saved together in a single table. At first glance that may seem unusual because each of those activities seems to be so different from the others within the Workflow Builder. The truth is they all share many common characteristics. For example, all of them can be included as a node in a process. All of them can return a result which can be modeled in a transition. And, all of them can be referenced as a process activity in a transition. The value found in the “type” field is used to differentiate the various types of activities. The “type” field and the “result_type” field each are the subject of an expanded explanation in the next couple of sections.

For function activities only, the field “function” is used to store the name of the PLSQL procedure that the Workflow Engine should call to implement the function. A major portion of this paper will be devoted to writing and understanding that procedure, so we will see this again, but it is the “function” field in this table that stores the association between the function activity that you see in the Workflow Builder and the PLSQL code that actually does the work. The discussion of writing code to implement function activities begins on page 30.

 © Copyright 2008 by Daniel Stober

For notification activities only, the field called “message” will be populated. In these cases, it will contain the internal name of the message that the notification will deliver. A notification activity is just a specialized activity that delivers a message to a role. When you create a notification, the Workflow Builder requires you to associate a message with that notification. It is in the “message” field that that association is stored.

Also worth noting here is the purpose of a couple of other fields. The “cost” field is the cost threshold for this activity that the Workflow Engine will apply to determine whether or not this activity should be deferred. The error_item_type and error_process together represent the process that the Workflow Engine will launch if an exception is raised while executing this particular activity. Later in the paper, we’ll examine the error handling model more closely. (See page 38.) Finally, the “version” field works in conjuction with the begin_date and end_date fields to support Workflow’s familiar and powerful process versioning functionality.

THE “TYPE” FIELD

As you can see in the figure 4 on page 5, within the Navigator Window in the Workflow Builder, there are seven sub-headings under each item_type: attributes, processes, function, notifications, events, messages and lookup types. When you save or modify your process in the database, the values from four of these subheadings are saved together in this single table, as illustrated in the figure. The “type” field is the way that the individual types of activities can be distinguished. There are five valid values found in the “type” field: “FUNCTION”, “NOTICE”, “EVENT”, “PROCESS”, and “FOLDER”. Those values should look familiar because they are identical or (in the case of “NOTICE”) similar to the subcategories from the Navigator Window.

The Folder is the encapsulating activity of a process. Each runnable process will have only one record with a type of “FOLDER” active at one time.

If you ever have tried to create a process and a function activity with the same internal name, you have seen that the Workflow Builder does not permit it. This is because the values in the “item_type” and “name” fields constitute the primary key on this table, so they must represent a unique combination.

PROCESS VERSIONS AND DATES

In this discussion, wf_activities is the first table we’ve met that touches on the concept of versions. You’re probably aware of one of the coolest things about Oracle Workflow: that it supports multiple versions of the same process running at the same time.

Say, for example, that you have an accounts payable process that seeks approval from the budget director before allowing an invoice to be paid. Later, you decide that you want to alter the process to require an approval from the comptroller for large transactions. You can modify your process and save the changes to the database, but your actions will not affect those processes that already are running. The previously launched processes retain the process definition that was in force at the time they were launched.

The version number works in concert with the “begin_date” and “end_date” fields, to ensure that only one version of any activity is active at any given time. (“End_date” is not a required field, so the last version will have a begin_date but no end_date, just something to keep in mind when you are writing queries.)

RESULT TYPE

If you intend to model transitions in a process based upon values returned by an activity node, then the expected results must be predefined by supplying a lookup type, which is stored in this field. We’ll cover lookup_types and lookup_codes in a little more depth below on page 14.

The result type stored is the internal name of the lookup.

WF_ACTIVITY_ATTRIBUTES
WF_ACTIVITY_ATTRIBUTES_TL
WF_ACTIVITY_ATTRIBUTES_VL

```
SQL> desc wf_activity_attributes
 Name                            Null?    Type
 ------------------------------- -------- ----
 ACTIVITY_ITEM_TYPE              NOT NULL VARCHAR2(8)
 ACTIVITY_NAME                   NOT NULL VARCHAR2(30)
 ACTIVITY_VERSION                NOT NULL NUMBER
 NAME                            NOT NULL VARCHAR2(30)
 SEQUENCE                        NOT NULL NUMBER
 TYPE                            NOT NULL VARCHAR2(8)
 VALUE_TYPE                      NOT NULL VARCHAR2(8)
 PROTECT_LEVEL                   NOT NULL NUMBER
 CUSTOM_LEVEL                    NOT NULL NUMBER
 SUBTYPE                                  VARCHAR2(8)
 FORMAT                                   VARCHAR2(240)
 TEXT_DEFAULT                             VARCHAR2(4000)
 NUMBER_DEFAULT                           NUMBER
 DATE_DEFAULT                             DATE
 SECURITY_GROUP_ID                        VARCHAR2(32)

SQL> desc wf_activity_attributes_tl
 Name                            Null?    Type
 ------------------------------- -------- ----
 ACTIVITY_ITEM_TYPE              NOT NULL VARCHAR2(8)
 ACTIVITY_NAME                   NOT NULL VARCHAR2(30)
 ACTIVITY_VERSION                NOT NULL NUMBER
 NAME                            NOT NULL VARCHAR2(30)
 LANGUAGE                        NOT NULL VARCHAR2(30)
 DISPLAY_NAME                    NOT NULL VARCHAR2(80)
 PROTECT_LEVEL                   NOT NULL NUMBER
 CUSTOM_LEVEL                    NOT NULL NUMBER
 DESCRIPTION                              VARCHAR2(240)
 SOURCE_LANG                     NOT NULL VARCHAR2(4)
 SECURITY_GROUP_ID                        VARCHAR2(32)
```

Earlier we saw that processes, functions, notifications, and events all are stored together in a single table. Similarly, the activity attributes associated with all of these activities also are stored in one table.

(By the way, notification activities can have activity attributes, too, but the attributes most often used with notification are message attributes, which are not stored in this table.)

Notice that the table requires three fields just to identify to which activity the attribute is attached: the item_type, name, and version of the activity. To join this table to the wf_activities tables you must join all three of these fields to their corresponding fields in that table. “Activity_item_type” from this table joins to “item_type” in the latter table, “activity_name” joins to “name”, and “activity version” to “version”.

Combining those three fields with the “name” field means that it takes distinct values in four different fields to constitute a unique record in this table.

Like we saw in wf_item_attributes, the “type” field is this table refers to the datatype of the values that the attribute will contain. Activity attributes differ from item_attributes in that workflow does not keep track of runtime values for them. The value of an activity attribute either must be set during the design phase in the process window of the Workflow Builder, or the value must be based on the runtime value of an item_attribute. The field “value_type” is where this information is stored, and that field will contain one of two values, “ITEMATTR” or “CONSTANT”.

WF_ACTIVITY_ATTR_VALUES

```
SQL> desc wf_activity_attr_values
 Name                            Null?    Type
 ------------------------------- -------- ----
 PROCESS_ACTIVITY_ID             NOT NULL FLOAT(*)
 NAME                            NOT NULL VARCHAR2(30)
 VALUE_TYPE                      NOT NULL VARCHAR2(8)
 PROTECT_LEVEL                   NOT NULL FLOAT(*)
 CUSTOM_LEVEL                    NOT NULL FLOAT(*)
 TEXT_VALUE                               VARCHAR2(4000)
 NUMBER_VALUE                             FLOAT(*)
 DATE_VALUE                               DATE
 SECURITY_GROUP_ID                        VARCHAR2(32)
```

The next table we'll examine is the table used to track values contained in activity attributes. This table is identical in purpose to wf_item_attribute_values except it holds values for activity attributes instead of item attributes. However, behind the scenes this table is much different than the one used for item attributes because activity attributes are not changed at runtime.

The interesting thing about this table is that it uses the process_activity_id to identify the activity to which the attribute is attached. The same activity can be inserted into a process more than one time, so the only way to uniquely identify the node to which this attribute is attached is to use the process_activity_id.

Once again, there are APIs available to read values from this table, but there are none to insert and update activity attribute values. Activity attribute values either are set by the creator of the Workflow during design time or are based upon item attribute values.

WF_MESSAGES

WF_MESSAGES_TL

WF_MESSAGES_VL

```
SQL> desc wf_messages
 Name                            Null?    Type
 ------------------------------- -------- ----
 TYPE                            NOT NULL VARCHAR2(8)
 NAME                            NOT NULL VARCHAR2(30)
 PROTECT_LEVEL                   NOT NULL FLOAT(*)
 CUSTOM_LEVEL                    NOT NULL FLOAT(*)
 DEFAULT_PRIORITY                         FLOAT(*)
 READ_ROLE                                VARCHAR2(320)
 WRITE_ROLE                               VARCHAR2(320)
 SECURITY_GROUP_ID                        VARCHAR2(32)

SQL> desc wf_messages_tl
 Name                            Null?    Type
 ------------------------------- -------- ----
 TYPE                            NOT NULL VARCHAR2(8)
 NAME                            NOT NULL VARCHAR2(30)
 LANGUAGE                        NOT NULL VARCHAR2(30)
 DISPLAY_NAME                    NOT NULL VARCHAR2(80)
 SUBJECT                         NOT NULL VARCHAR2(240)
 PROTECT_LEVEL                   NOT NULL FLOAT(*)
 CUSTOM_LEVEL                    NOT NULL FLOAT(*)
 DESCRIPTION                              VARCHAR2(240)
 BODY                                     VARCHAR2(4000)
 SOURCE_LANG                     NOT NULL VARCHAR2(4)
 HTML_BODY                                VARCHAR2(4000)
 SECURITY_GROUP_ID                        VARCHAR2(32)
```

One of the types of records stored in th wf_activities tables is the notification, which is unique among the activities because it is associated with a message. The messages that are associated with those notifications are stored in this table.

As you examine these tables, bear in mind the localizations scheme discussed earlier. Each message, which is uniquely identified by the combination of item_type and message_name (stored in the fields "type" and "name") receives a single record in the wf_messages table. But a close inspection at the description of the wf_messages table reveals that it does *not* have a field containing the message text.

The text of the message is stored only in the localization table. Think back for a minute about the hypothetical situation outlined above on page 4 in the discussion about how wf_messages, wf_messages_tl, and wf_messages_vl interact. The company has a single message to send to customers about confirmed orders. But instead of creating different messages and notifications nodes for its Anglophone and Francophone patrons, the company can use a single Workflow process, with a single message, and Oracle Workflow will supply the correct message texts based upon language. So, the message texts are stored only in the wf_messages_tl table. They can found in the "body" and "html_body" fields.

 © Copyright 2008 by Daniel Stober

WF_MESSAGE_ATTRIBUTES

WF_MESSAGE_ATTRIBUTES _TL

WF_MESSAGE_ATTRIBUTES _VL

As the name suggests, message attributes are tied to a message. Those attributes can be used as tokens in the subject or body of a message template to place variables values into the message at runtime.

Message attributes function nearly identically to the way that activity attributes work. You cannot supply values for a message attribute at runtime, the values of these attributes are set to a constant value during design or are derived from the value of an item attribute at runtime.

```
SQL> desc wf_message_attributes
 Name                            Null?    Type
 ------------------------------- -------- ----
 MESSAGE_TYPE                    NOT NULL VARCHAR2(8)
 MESSAGE_NAME                    NOT NULL VARCHAR2(30)
 NAME                            NOT NULL VARCHAR2(30)
 SEQUENCE                        NOT NULL NUMBER
 TYPE                            NOT NULL VARCHAR2(8)
 SUBTYPE                         NOT NULL VARCHAR2(8)
 VALUE_TYPE                      NOT NULL VARCHAR2(8)
 PROTECT_LEVEL                   NOT NULL NUMBER
 CUSTOM_LEVEL                    NOT NULL NUMBER
 FORMAT                                   VARCHAR2(240)
 TEXT_DEFAULT                             VARCHAR2(4000)
 NUMBER_DEFAULT                           NUMBER
 DATE_DEFAULT                             DATE
 ATTACH                                   VARCHAR2(1)
 SECURITY_GROUP_ID                        VARCHAR2(32)

SQL> desc wf_message_attributes_tl
 Name                            Null?    Type
 ------------------------------- -------- ----
 MESSAGE_TYPE                    NOT NULL VARCHAR2(8)
 MESSAGE_NAME                    NOT NULL VARCHAR2(30)
 NAME                            NOT NULL VARCHAR2(30)
 LANGUAGE                        NOT NULL VARCHAR2(30)
 DISPLAY_NAME                    NOT NULL VARCHAR2(80)
 PROTECT_LEVEL                   NOT NULL NUMBER
 CUSTOM_LEVEL                    NOT NULL NUMBER
 DESCRIPTION                              VARCHAR2(240)
 SOURCE_LANG                     NOT NULL VARCHAR2(4)
 SECURITY_GROUP_ID                        VARCHAR2(32)
```

WF_PROCESS_ACTIVITIES

A process is a sequence of activities performed in a pre-determined order based upon defined transitions. Oracle Workflow uses two tables to represent this model. The first of the two, wf_process_activities, is the table that actually establishes the association of activities within processes. When you create a process definition in the Workflow Builder by dragging various notifications and functions into the process window, the records created by the Builder are stored into this table. The order in which the activities should be executed is stored in the second of the two tables, and that we'll see that one in the next section.

```
SQL> desc wf_process_activities
Name                            Null?    Type
------------------------------- -------- ----
PROCESS_ITEM_TYPE               NOT NULL VARCHAR2(8)
PROCESS_NAME                    NOT NULL VARCHAR2(30)
PROCESS_VERSION                 NOT NULL NUMBER
ACTIVITY_ITEM_TYPE              NOT NULL VARCHAR2(8)
ACTIVITY_NAME                   NOT NULL VARCHAR2(30)
INSTANCE_ID                     NOT NULL NUMBER
INSTANCE_LABEL                  NOT NULL VARCHAR2(30)
PERFORM_ROLE_TYPE               NOT NULL VARCHAR2(8)
PROTECT_LEVEL                   NOT NULL NUMBER
CUSTOM_LEVEL                    NOT NULL NUMBER
START_END                                VARCHAR2(8)
DEFAULT_RESULT                           VARCHAR2(30)
ICON_GEOMETRY                            VARCHAR2(2000)
PERFORM_ROLE                             VARCHAR2(320)
USER_COMMENT                             VARCHAR2(240)
SECURITY_GROUP_ID                        VARCHAR2(32)
```

Each of the activities that comprise the process is represented by one record in the table. For a single record representing an activity in a process, there are five fields necessary to constitute each one: Three to pinpoint the process (and version of the process) and two more to identify the activity. Also on those records, you'll find the instance_label, and instance_id, the latter of which is the primary key on the table.

Additionally, there is one more record representing the process as a whole. On that one, the word "ROOT" appears in the process field, and the name of the process in the activity field.

Refer back to the discussion about wf_activities above, and recall that wf_activities stores both processes as well as function and notification activities. But a process itself is made up of function and notification activities. So, since the wf_activities tables stores both the process container and the process contents at the same time, it makes for some confusing joins in from this table.

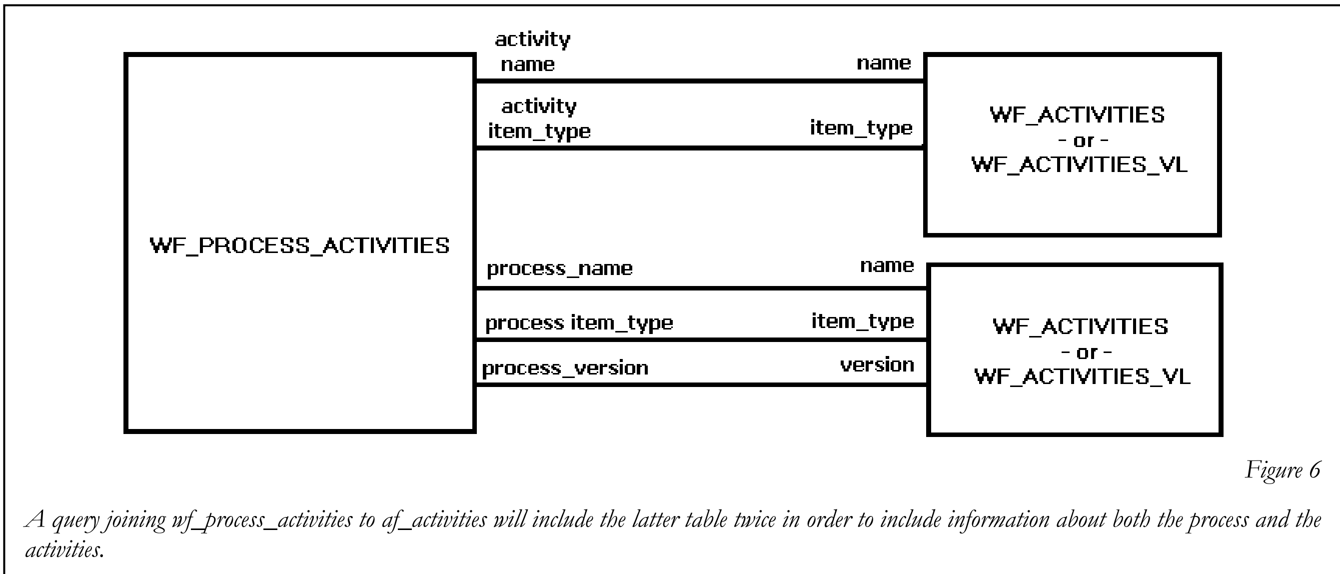

Figure 6

A query joining wf_process_activities to af_activities will include the latter table twice in order to include information about both the process and the activities.

Since processes are stored in the same tables with the activities that comprise the process, you'll have to include that table twice in any query that seeks to list all of the activities in a process. That makes for some interesting joins that can make a basic query pretty complicated. For example, in the query I mentioned, just to link the two tables (in three instances) that would appear in that query would require five joins. The chart featured above in figure 6 demonstrates all of those joins.

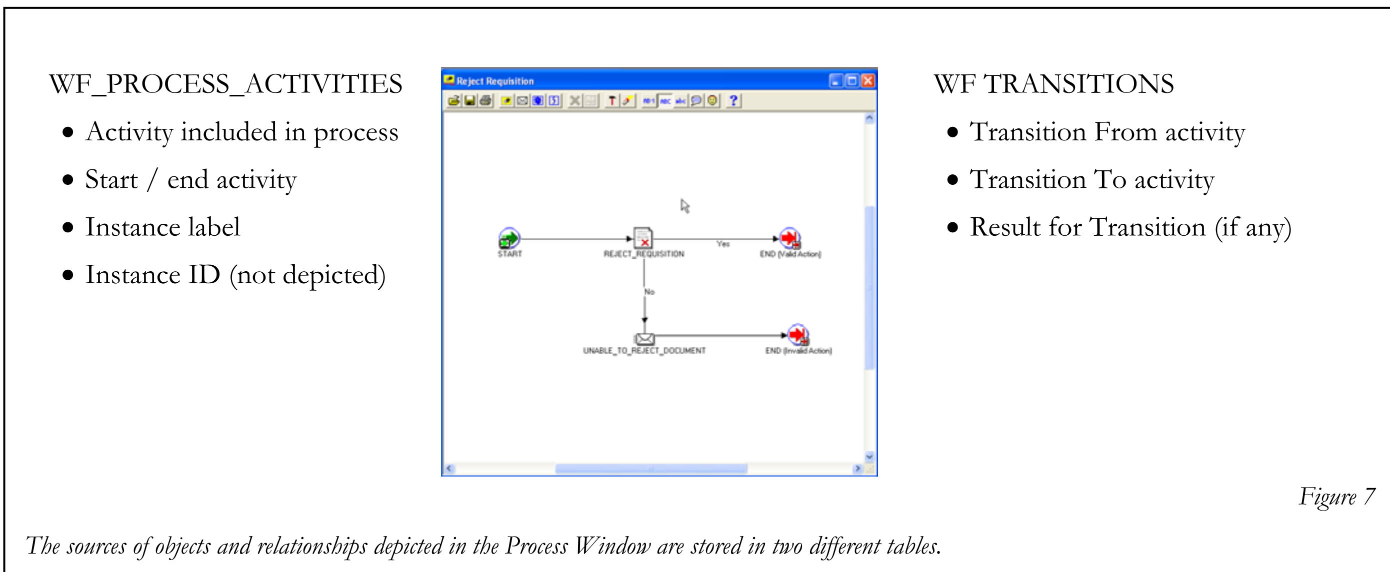

Figure 7

The sources of objects and relationships depicted in the Process Window are stored in two different tables.

In general, within Workflow, primary key constraints and foreign key relationships are established using fields containing varchar data. For example, joins between many tables are made with fields like item_type and item_key, both of them varchar. But, wf_process_activities is one of the few places in Workflow where Oracle uses a sequence number as the primary key. Each record in this table is defined by a unique key, instance_id. Among other things, that number is used to join to the transitions table. Most importantly, it is one of the parameter values supplied in function calls as we'll see in the coming sections when we begin the exploration into writing PLSQL.

 © Copyright 2008 by Daniel Stober

Making any changes to the process definition causes a new set of records to be inserted into this table, all of them with a new version. This is how Oracle Workflow implements the concept of version control. Any change to any of the activities in the process or to the process itself results in the creation of a complete set of records in the process. All of those records are tied together by virtue of having the same in the version number – each new set of records created contains the next highest version number.

When we get to the wf_items table, we'll see that whenever a Workflow process is launched, the number of the version which is in effect at the time of the launching is stored in wf_items. Later changes to the process will not impact the items which are already running because they will continue to run under the version in place when they were launched.

WF_ACTIVITY_TRANSITIONS

```
SQL> desc wf_activity_transitions
 Name                            Null?    Type
 ------------------------------- -------- ----
 FROM_PROCESS_ACTIVITY           NOT NULL NUMBER
 RESULT_CODE                     NOT NULL VARCHAR2(30)
 TO_PROCESS_ACTIVITY             NOT NULL NUMBER
 PROTECT_LEVEL                   NOT NULL NUMBER
 CUSTOM_LEVEL                    NOT NULL NUMBER
 ARROW_GEOMETRY                           VARCHAR2(2000)
 SECURITY_GROUP_ID                        VARCHAR2(32)
```

The one piece of information from the process design window that is not saved in the wf_process_activities table is the flow of the process from node to node as indicated by the transition arrows. That information is stored in this table.

A transition is defined by three discrete pieces of information: the node where the arrow begins, the node toward which the arrow points, and the result which, when returned by the beginning node, causes the transition to be followed. Not surprisingly, it is those three fields which are the most important fields in this table: “from_process_activity”, “to_process_activity”, and “result_code”.

The values stored in “from_process_activity” and “to_process_activity” are numbers which represent the instance_id of the records from wf_process_activities from which and to which the transition is moving. That takes care of two of the three essential elements of a transition. The third component comes into play for transitions away from process_activities that return a result. That pieced of the equation is stored in the “result_code” field.

As an example, in a simple approval process, there likely would be two different flow transitions heading away from an approval notification. The two outbound transitions departing the notification node would be represented by two records in the wf_activity_transitions table. Both records would reference the instance_id of the approval notification node in the “from_process_activity” field. But, one of the records would have a value like “APPROVED” in “result_code” and the instance_id of the next node on that branch of the flow, and the other record might contain “DENIED” and the appropriate instance_id for that flow's next activity.

The result_code should be a valid member of the lookup_type that is defined for the activity, as it is stored in wf_activities. In the example above, the process designer would have had to create a lookup type that included “APPROVED” and “DENIED” and then would have had to specify that lookup type as the result type for that activity. We'll explore wf_lookup_types and wf_lookups in the next paragraphs.

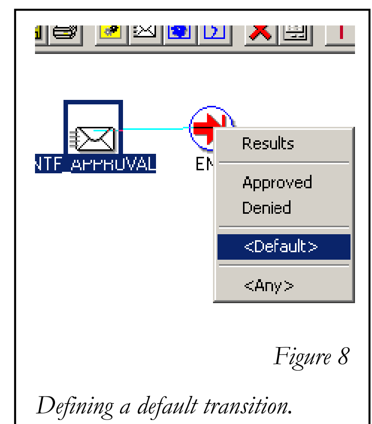

Figure 8

Defining a default transition.

If a process activity returns a result that is not modeled for a transition, the Workflow engine does not have a path available to follow to continue the flow, and the process activity becomes #STUCK. Put another way, not modeled means that there are no records in the wf_activity_transitions table matching the values of the current activity's instance_id and the actvity's result in the “from_process_activity” and “result_code” fields respectively.

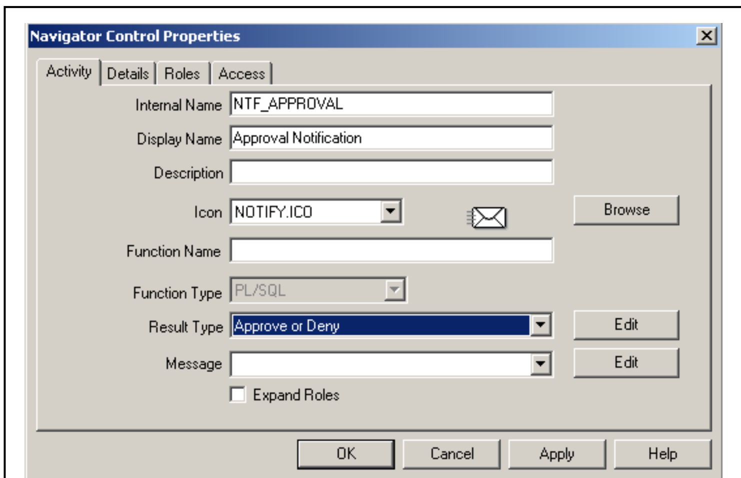

Figure 9

Choosing a result type for an activity. The values of the lookup selected here become the valid result_codes to be used by wf_transitions

The way to prevent this possibility is to assign a flow to be used as a default. In cases where a default alternative is designated, the transition record contains an asterisk ("*") in the "result_code" field. The default tells the Workflow engine to follow the designated flow in the event that the result returned by the process activity is not explicitly modeled.

WF_LOOKUP_TYPES_TL

WF_LOOKUP_TYPES

Wf_lookup types_tl is the table used to set up the types of results expected from Workflow activities like functions and notifications. This table does *not* contain the actual result values, it holds the groupings of the result_codes – the names you see in the Workflow Builder as the names of the Lookups.

```
SQL> desc wf_lookup_types_tl
 Name                            Null?    Type
 ------------------------------- -------- ----
 LOOKUP_TYPE                     NOT NULL VARCHAR2(30)
 DISPLAY_NAME                    NOT NULL VARCHAR2(80)
 LANGUAGE                        NOT NULL VARCHAR2(30)
 ITEM_TYPE                       NOT NULL VARCHAR2(8)
 PROTECT_LEVEL                   NOT NULL NUMBER
 CUSTOM_LEVEL                    NOT NULL NUMBER
 DESCRIPTION                              VARCHAR2(240)
 SOURCE_LANG                     NOT NULL VARCHAR2(4)
 SECURITY_GROUP_ID                        VARCHAR2(32)
```

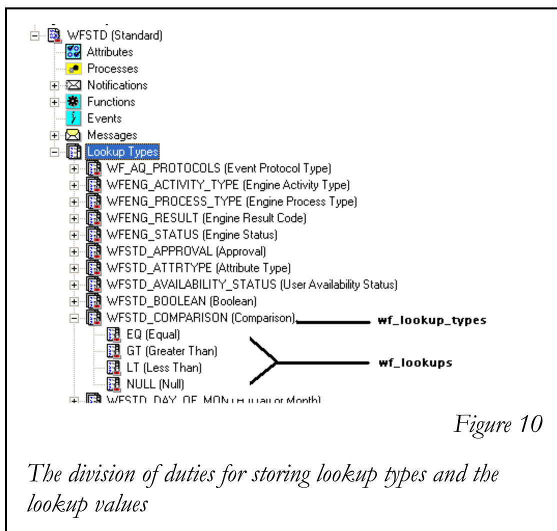

Figure 10

The division of duties for storing lookup types and the lookup values

For example, consider the "Comparison" lookup type that is found in the Standard item_type. The single record that constitutes the lookup type is stored in this table. The values that comprise this lookup are stored in the table we will meet next, and are joined together by the common value they share in the "lookup_type" field. See Fig 10 at left.

An additional noteworthy item is that the two lookups tables do not follow the same pattern to support localizations that the other setup tables employ. In the case of these two tables, the object name ending in "_TL" is the base table, and the object name with no suffix is the translation view.

WF_LOOKUPS_TL

WF_LOOKUPS

```
SQL> desc wf_lookups_tl
 Name                            Null?    Type
 ------------------------------- -------- ----
 LOOKUP_TYPE                     NOT NULL VARCHAR2(30)
 LOOKUP_CODE                     NOT NULL VARCHAR2(30)
 MEANING                         NOT NULL VARCHAR2(80)
 LANGUAGE                        NOT NULL VARCHAR2(30)
 PROTECT_LEVEL                   NOT NULL NUMBER
 CUSTOM_LEVEL                    NOT NULL NUMBER
 DESCRIPTION                              VARCHAR2(240)
 SOURCE_LANG                     NOT NULL VARCHAR2(4)
 SECURITY_GROUP_ID                        VARCHAR2(32)
```

Wf_lookups_tl is the table that stores the component values that comprise a lookup_type. Continuing the example begun in the prior section, wf_lookups_tl is the table that holds each of the four possible lookup values that make up the Comparison lookup type. This table joins to wf_lookup_types using the "lookup_type" field. Again, like we saw with wf_lookup_types_tl, it is the "TL" table that is the base table here, too.

Chapter 2: THE RUNTIME TABLES

WF_ITEMS

WF_ITEMS_V

```
SQL> desc wf_items
 Name                            Null?    Type
 ------------------------------- -------- ----
 ITEM_TYPE                       NOT NULL VARCHAR2(8)
 ITEM_KEY                        NOT NULL VARCHAR2(240)
 ROOT_ACTIVITY                   NOT NULL VARCHAR2(30)
 ROOT_ACTIVITY_VERSION           NOT NULL FLOAT(*)
 OWNER_ROLE                               VARCHAR2(320)
 PARENT_ITEM_TYPE                         VARCHAR2(8)
 PARENT_ITEM_KEY                          VARCHAR2(240)
 PARENT_CONTEXT                           VARCHAR2(2000)
 BEGIN_DATE                      NOT NULL DATE
 END_DATE                                 DATE
 USER_KEY                                 VARCHAR2(240)
 HA_MIGRATION_FLAG                        VARCHAR2(1)
 SECURITY_GROUP_ID                        VARCHAR2(32)
```

This is the primary table used to track Workflow process instances. Just as wf_item_types is at the top of the hierarchy of the definition tables, wf_items is the basis for the runtime tables. All of the other runtime tables can be joined to a record in this table, either directly or indirectly. A unique instance of a process running for a particular item_type is identified by a unique item_key , and that unique combination of item_type and item_key constitutes a unique record in this table. Records in this table can be linked back to the definition tables using item_type.

When a new process is launched, the name of the top level process and the number of the version in place at the time are stored in the fields "root_activity" and "root_activity_version." The "begin_date" field contains the date and time that the process was launched, and "end_date" is populated for those items that have been completed.

Finally, notice the presence of the fields "parent_item_type" and "parent_item_key". These fields are populated in those items which are created as the result of another process. When you see values in this field, they are an indication either of a sub-process or of an error process.

For security purposes, Oracle provides a SELECT only view of this table. I have noted the existence of this view in parentheses above, wf_items_v. This is not a view related to localizations.

WF_ITEM_ACTIVITY_STATUSES

WF_ITEM_ACTIVITY_STATUSES _V

```
SQL> desc wf_item_activity_statuses
 Name                            Null?    Type
 ------------------------------- -------- ----
 ITEM_TYPE                       NOT NULL VARCHAR2(8)
 ITEM_KEY                        NOT NULL VARCHAR2(240)
 PROCESS_ACTIVITY                NOT NULL NUMBER
 ACTIVITY_STATUS                          VARCHAR2(8)
 ACTIVITY_RESULT_CODE                     VARCHAR2(30)
 ASSIGNED_USER                            VARCHAR2(320)
 NOTIFICATION_ID                          NUMBER
 BEGIN_DATE                               DATE
 END_DATE                                 DATE
 EXECUTION_TIME                           NUMBER
 ERROR_NAME                               VARCHAR2(30)
 ERROR_MESSAGE                            VARCHAR2(2000)
 ERROR_STACK                              VARCHAR2(4000)
 OUTBOUND_QUEUE_ID                        RAW(16)
 DUE_DATE                                 DATE
 SECURITY_GROUP_ID                        VARCHAR2(32)
 ACTION                                   VARCHAR2(30)
 PERFORMED_BY                             VARCHAR2(320)
```

This table tracks the progression of a process instance through the various activity nodes in the process definition. Each activity from the process definition is given one entry in this table as the process transitions to it. The Workflow Engine tracks the activity by changing the value of the status for each of the activities in turn as they are completed. For querying, Oracle provides a SELECT only version of this table in the form of the view wf_item_activity_statuses_v.

For a process that is active and running, this table holds the key to the entire engine. In this table, you can see the activities have completed and the activities that are still in progress, and by joining back to the definition table wf_activities, you can discover exactly what those activities are. The field “process activity” provides a link back to the definition tables because it joins to wf_process_activites on the field “instance_id”. From there, you can join to wf_activities to find the names both of the activity and of the process.

The primary_key on this table is comprised of the three not null fields, “item_type,” “item_key,” and “process_activity.” Since a primary_key must consist of a unique value, the effect of this is that any single activity in a process instance can only appear in this table one time. If process flow happens to revisit the same activity node more than once – for example, due to a loop – only one record of that occurrence can be retained in this table. In such scenarios, the record from the prior execution is copied into the history table, wf_item_activity_statuses_h, and the results from the last visit are recorded here.

Another key link to this table is wf_activity_transitions. When an activity is completed, the Workflow engine uses the combination of the values from “process_activity” and “activity_result_code” to see what activity or activities are next up.

The field “activity_status” is the place you should check in order to understand exactly what happened – or is happening – within a particular activity. The two most common statuses are “ACTIVE”, which means that processing in the activity is underway, and “COMPLETE”, which tells you that the activity finished successfully. An activity waiting at an AND join will have a status of “WAITING”. See the list at right for the complete list of possible activity statuses. All of the statuses are defined as constants in the package wf_engine, so, for example, “WAITING” can be referenced in PLSQL code as wf_engine.waiting as an alternative to hard-coded strings, but the value stored in the table is the string itself.

Values in “activity_status”

COMPLETE	Normal completion
ACTIVE	Activity running
WAITING	Activity is waiting to run
NOTIFIED	Notification delivered and open
SUSPEND	Activity suspended
DEFERRED	Processing deferred
ERROR	Completed with error

Figure 11

The possible values that can be found in the “activity_status” field and their meanings. These values appear again in the section on resultOut on page 34.

For activities that have completed, check the value in “activity_result_code”. This field holds the result that was returned by the code that implemented it. That’s the technical way of saying that this field holds the value that was returned by whatever procedure the activity at this node called. For example, a function that answers a Boolean question, might be set up to return a value from “Standard Yes/No”. Once this function completes and returns its result of “Y” or “N”, the Workflow Engine stores that result in the activity_result_code field on the record that called the function.

In the query example shown in Figure 12, you can see the values for one process instance as it is stored in the table. All of the activities listed were completed successfully. The first two activities did not return any result, so the Workflow engine placed the

value “#NULL” into the “activity_result_code” column. The third activity resulted in “MYRESULT” and the last one, “EXIT.” Since the table contains a field for process_activity, it’s easy enough to join back to wf_process_instances and then to wf_activities to discover exactly what activity each of these is. Although I did not perform the join here, the first record, where process_activity is 119085, happens to be the record for the process as a whole.

```
  1  SELECT * FROM wf_item_activity_statuses
  2  WHERE item_type = 'DJS_DEMO'
  3* AND item_key    = '140938'
SQL> /

ITEM TYP ITEM KEY             PROCESS ACTIVITY ACTIVITY ACTIVITY RESULT CODE
-------- -------------------- ---------------- -------- --------------------
DJS_DEMO 140938                         119085 COMPLETE #NULL
DJS_DEMO 140938                         119183 COMPLETE #NULL
DJS_DEMO 140938                         119185 COMPLETE MYRESULT
DJS_DEMO 140938                         119187 COMPLETE EXIT

4 rows selected.
```

Figure 12

There’s one more thing to note: this table contains a “notification_id” field. This field is populated with a value only for notification activities; otherwise, it will be blank. It is this field that permits you to join to wf_notifications. This nexus actually is more useful to know when writing queries to move in the other direction: In other words, given a notification_id, you can join to this table to get the item_key associated with that notification.

GOTCHA: #NULL is not the same thing as NULL.

When you see the value #NULL in the field activity_result_code, it is not the same thing as a null value. The value with leading hash character tells you that the activity was completed, but the function activity did not return any value. An actual null in the field generally means that the activity has not yet completed.

WF_ITEM_ACTIVITY_STATUSES_H

This is a special table used to track status histories of activities that have been revisited. The “_h” in the table name is short for “history”. Since each activity node in a process definition can have only one record per item_key in the wf_item_activity_statuses table, when the flow in a process instance returns to an activity that already has a record in the wf_item_activity_statuses table, a record is created here in the history table. The Workflow Engine creates a copy of that existing record from wf_item_activity_statuses and places it into this table, then deletes that record from the former table, in order to make way for the latest record from the new activity in the main wf_item_activity_statuses table.

```
SQL> desc wf_item_activity_statuses_h
 Name                            Null?    Type
 ------------------------------- -------- ----
 ITEM_TYPE                       NOT NULL VARCHAR2(8)
 ITEM_KEY                        NOT NULL VARCHAR2(240)
 PROCESS_ACTIVITY                NOT NULL NUMBER
 ACTIVITY_STATUS                          VARCHAR2(8)
 ACTIVITY_RESULT_CODE                     VARCHAR2(30)
 ASSIGNED_USER                            VARCHAR2(320)
 NOTIFICATION_ID                          NUMBER
 BEGIN_DATE                               DATE
 END_DATE                                 DATE
 EXECUTION_TIME                           NUMBER
 ERROR_NAME                               VARCHAR2(30)
 ERROR_MESSAGE                            VARCHAR2(2000)
 ERROR_STACK                              VARCHAR2(4000)
 OUTBOUND_QUEUE_ID                        RAW(16)
 DUE_DATE                                 DATE
 SECURITY_GROUP_ID                        VARCHAR2(32)
 ACTION                                   VARCHAR2(30)
 PERFORMED_BY                             VARCHAR2(320)
```

This table contains precisely the same columns as wf_item_activity_statuses, so that you can UNION it with the other table in queries where you need a complete picture of all of the activities that occurred in a process instance.

WF_NOTIFICATIONS

```
SQL> desc wf_notifications
 Name                            Null?    Type
 ------------------------------- -------- ----
 NOTIFICATION_ID                 NOT NULL FLOAT(*)
 GROUP_ID                        NOT NULL FLOAT(*)
 MESSAGE_TYPE                    NOT NULL VARCHAR2(8)
 MESSAGE_NAME                    NOT NULL VARCHAR2(30)
 RECIPIENT_ROLE                  NOT NULL VARCHAR2(320)
 STATUS                          NOT NULL VARCHAR2(8)
 ACCESS_KEY                      NOT NULL VARCHAR2(80)
 MAIL_STATUS                              VARCHAR2(8)
 PRIORITY                                 FLOAT(*)
 BEGIN_DATE                               DATE
 END_DATE                                 DATE
 DUE_DATE                                 DATE
 RESPONDER                                VARCHAR2(320)
 USER_COMMENT                             VARCHAR2(4000)
 CALLBACK                                 VARCHAR2(240)
 CONTEXT                                  VARCHAR2(2000)
 ORIGINAL_RECIPIENT              NOT NULL VARCHAR2(320)
 FROM_USER                                VARCHAR2(320)
 TO_USER                                  VARCHAR2(320)
 SUBJECT                                  VARCHAR2(2000)
 LANGUAGE                                 VARCHAR2(4)
 MORE_INFO_ROLE                           VARCHAR2(320)
 FROM_ROLE                                VARCHAR2(320)
 SECURITY_GROUP_ID                        VARCHAR2(32)
```

This is the table used to track notifications. When the Workflow engine creates a new notification, it adds a record to this table.

The key fields are in this table are the two status fields, “status” and “mail_status”. The field “status” conveys the status of the overall notification, and will contain one of three values: “OPEN”, “CLOSED”, or “CANCELED.”

Since this table does not contain a field for item_key, you must use the notification_id to join to wf_item_activity statuses (or wf_item_activity_statuses_h) to determine that datum.

When troubleshooting issues with the Notification Mailer, you can check the status of that action in the “mail_status” column. A value of “MAIL” indicates that the notification is waiting for the Mailer to recognize and handle the notification sent event and deliver the notification to the email system. “SENT” indicates that the Mailer operation was successful, while “ERROR” and “FAILED” indicate otherwise. Among my most frequent uses for this table is a query on the “mail_status” field when troubleshooting Notification Mailer issues.

WF_ITEM_ATTRIBUTE_VALUES

```
SQL> desc wf_item_attribute_values
 Name                            Null?    Type
 ------------------------------- -------- ----
 ITEM_TYPE                       NOT NULL VARCHAR2(8)
 ITEM_KEY                        NOT NULL VARCHAR2(240)
 NAME                            NOT NULL VARCHAR2(30)
 TEXT_VALUE                               VARCHAR2(4000)
 NUMBER_VALUE                             FLOAT(*)
 DATE_VALUE                               DATE
 EVENT_VALUE                              UNDEFINED
 SECURITY_GROUP_ID                        VARCHAR2(32)
```

This table holds the values placed into the attributes of running process. Note the three NOT NULL fields in the table. “Item_key” and “item_type” are self-documenting, while the “Name” field contains the internal_name of the attribute whose value is contained in this particular record.

The next three fields are where the work of this table is done. “Text_value”, “number_value”, and “date_value” are the fields that store actual value contained in the attribute. One and *only one* of those fields will contain a value in any given record, but which one depends on the datatype of the attribute whose value is being stored. You may recall from the discussion given on page 8 that the datatype of the attribute is stored in the “type” field in wf_item_attributes.

Remember, this is the only runtime table for tracking any attribute values. Message attributes and activity attributes do not change for specific processes; they are based either on the value from an item attribute or from a constant, so that information is stored in the respective definition table.

Of course, in order to update value the value of an item attribute you’ll need to use Oracle’s APIs, because you should never update application tables with your own code. You’ll also want to use those APIs to read values from this table when writing PLSQL programs, because they employ caching for efficiency. There are three different APIs for each task, depending upon the datatype. Those will be explored in the API section, beginning on page 26.

GOTCHA: Two of the three fields always will be NULL.

Those three separate fields used to store values can snag you if you are unaware of how the data is stored. Say for example, that you needed to write a query looking for wf_item_attribute_values records that are null. Since two of the three fields are guaranteed to have NULL value, when writing such a query, you have to test all three fields for nulls:

```
WHERE text_value   IS NULL
  AND number_value IS NULL
  AND date_value   IS NULL
```

UNIT II: ORACLE'S DELIVERED SCRIPTS AND APIs

Chapter 3: WORKFLOW SCRIPTS

With the installation of Workflow, Oracle provides a number of scripts for you to use to check on the status of Workflow definitions and process instances, and to correct possible problems encountered. Many of the scripts perform tasks that are not handled by the Workflow Builder or any of the Workflow APIs, such as cleanup tasks like deleting attributes from item types.

Additionally, the scripts provide a valuable resource for someone who is trying to learn the Workflow application. I found these scripts to be very useful in helping to develop an understanding of the workings of Workflow. I dissected and studied the queries in the scripts to acquire an insight into how the data are stored and how the tables are interrelated.

The scripts can be found on the server where your database is installed. For the standalone cartridge version of Workflow, those scripts were located in a different directory than the one used for E-Business Suite installations. But since the standalone version was desupported in 2006, we need only by concerned with telling you where to find the scripts in your E-Business Suite installation:

```
$FND_TOP/sql
```

CAVEAT EXECUTOR

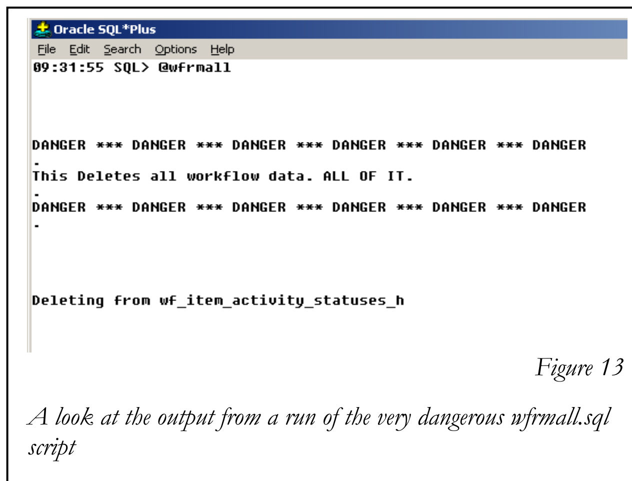

Figure 13

A look at the output from a run of the very dangerous wfrmall.sql script

Before we begin our look at Oracle's delivered scripts, I offer this caveat: Be careful when running scripts. Make sure that you understand what a script does before you run it. If you're not sure, then don't run it until you are.

I cannot overstate this warning. Let me show you an example. In the directory containing the delivered scripts, you'll find a script called wfrmall.sql sitting nondescriptly amidst the others. By outward appearances, it seems to be the same as those others. But this script is far more powerful than the others. "Wfrmall" stands for "Workflow remove all," and if you run this one, it will delete *all* of the Workflow data from *all* of the Workflow tables in your database. That's *all* of the data in *all* of the runtime tables and *all* of the data in *all* of the definition tables. Then, just to make sure that those deletions truly are final, the script includes all of the necessary COMMIT statements, so there is no turning back. If you end up running this one by mistake, you'll have to completely reinstall Workflow, and all of your runtime process information will be gone.

At least Oracle's developers have included some last chance warnings in the remarks at the top of the script, trying to impress upon the user just how serious this script is. Take a took at the screenshot in figure 13.

The bottom line is this: The scripts are very useful, but they are so useful because they're so powerful. Make sure that you know what a script does before you run it. Read the documentation. Review the code itself. Test it in a development environment.

REPORTING SCRIPTS

To begin we'll look at some scripts that query your Workflow data, providing diagnostic information in a report format. The reporting scripts are benign because they do not make any changes to the data. You can feel free to run any of these without fear.

wfstatus.sql

Wfstatus.sql is one of the most handy of these scripts. This script uses six queries to provide a quick overview of everything that has happened in a single Workflow process. It prompts for an item_type and item_key, and reports on that item's record in wf_items, followed by a query of wf_item_activity_statuses for a report on the progress in all of the activities in the item. Then, there are three queries for activity errors; the first shows the error stack and error message for any process activities that ended in error, and the second queries the activity_statuses for the process which was spawned by the error, and the returns the error stack and error message for any activities in the error process that errored out themselves. Finally, the last query is a query of the wf_item_attribute_values for the item_type and item_key.

Following this paragraph is an example of the output from a sample run of wfstatus.sql against a process that is stuck with an error:

```
16:57:53 SQL> @wfstatus
Enter value for 1: DJS_DEMO
Enter value for 2: 165655
**** WorkFlow Item

Item Type  Item Key        Begin Date         End Date           Activity
---------- --------------- ------------------ ------------------ ------------------------
DJS_DEMO   165655          17-FEB-08 16:56:55                    FEEDBACK

1 row selected.

**** Activity Statuses

Begin Date         Activity                                      Status   Result          User
------------------ --------------------------------------------- -------- ---------------
17-FEB-08 16:56:55 ROOT/Feedback Procedure                       ACTIVE   #NULL
17-FEB-08 16:56:55 Feedback Procedure/Start                      COMPLETE #NULL
17-FEB-08 16:56:55 Feedback Procedure/Dans PLSQL Procedure Call  ERROR    #EXCEPTION

3 rows selected.

**** Errored Activities

Activity                                      Result          Error Name
--------------------------------------------- --------------- ---------------------------
Error Message
------------------------------------------------------------------------------------------
Error Stack
------------------------------------------------------------------------------------------
Dans PLSQL Procedure Call                     #EXCEPTION      -1422
ORA-01422: exact fetch returns more than requested number of rows



1 row selected.

*** Error Process Activity Statuses
```

```
Begin Date         Activity                                      Status   Result          User
------------------ --------------------------------------------- -------- --------------- ------------
---
17-FEB-08 16:56:56 ROOT/Default Error Process                    ACTIVE   #NULL
17-FEB-08 16:56:56 Default Error Process/Start                   COMPLETE #NULL
17-FEB-08 16:56:56 Default Error Process/Initialize Error        COMPLETE COMPLETE
17-FEB-08 16:56:56 Default Error Process/Notify Administrator    NOTIFIED                 SYSADMIN

4 rows selected.

**** Error Process Errored Activities

no rows selected

**** Attribute Values

Attribute Name                 Value
------------------------------ ------------------------------
FEEDBACK_A
.MONITOR_KEY                   2849984401
.ADMIN_KEY                     256084351
#SCHEMA                        APPS

4 rows selected.


Commit complete.

16:58:36 SQL>
```

By the way, you may have noticed that the last line in the output references a COMMIT. Don't worry about that; there are no DML statements in this script.

wfverchk.sql

Use the wfverchk.sql script to find and identify activities with overlapping active date ranges. For any activity or process, only one version can be active at any given time. This script identifies data corruption in either of two scenarios:

If end_date is not null and the end_date is after the begin_date of the activity with the next-lowest begin_date

- Or-

If end-date IS null and any other record with a begin_date greater than the begin_date of this record

If the script identifies any problems, then you can use wfverupd.sql to fix them. More information on that is available in the section below on scripts that modify data.

Wfprotck.sql

Find Workflow items with improper values for customization levels or protection levels.

Use wfprot.sql to fix those items.

wfbkgchk.sql

Lists all items awaiting a run of the background engine. This includes activities with a status of 'DEFERRED' or 'WAIT', and items which have timed out.

DATA CLEANUP SCRIPTS

wfverupd.sql

Use this script to correct problems with versions and overlapping date ranges as identified by wfverchk.sql. In cases of conflicts, it will update the wf_activities table to set the one with the earlier begin date to have an end date equal to the next begin_date – thus eliminating the conflict.

wfrmitt.sql

This script eliminates all traces of an item type including all process instances, even if they are still active. Be careful, because the script is very powerful, and, like the others that delete data, it includes commit statements.

Use this script to retire an item_type from your installation, or if you decide that you need to erase the item_type and re-install it. If you are just trying to rebuild the item_type, remember that this script removes the active processes, too, and they cannot be rebuilt.

wfrmita.sql

Use this script when you've decided to eliminate an item attribute from a Workflow process. The script removes runtime data from wf_item_attribute_values and definition records from wf_item_attributes.

Before deleting an activity, check to see whether there are any active process which still use the attribute, because the script deletes records from wf_item_attribute_values without first checking to see if the records being removed are referenced in an active process.

You can use the following query to help identify those active processes:

```
select *
  from wf_item_attribute_values wiav
 where name     = '&attribute_name'
  and item_type = '&item_type'
  and item_key in (select item_key
                      from wf_items wi
                    where wiav.item_key  = wi.item_key
                         and wiav.item_type = wi.item_type
          and wi.end_date is not null)
```

wfchact.sql

Change the internal name of an activity and all references to it. One thing to note, however, is that it does not change instance labels, which by default are based upon the internal name of the activity when it is first added into a process. You'll have to update those yourself in the Workflow Builder.

wfchitt.sql

This script will change the internal name of an item type and all references to it. The beauty of this script -- aside from its obvious use – is that it makes changes in virtually every workflow table, so studying it is very instructive for learning to understand the data model.

wfchacta.sql

wfchita.sql

These scripts will assign a new internal name to an attribute. Use the first one, wfchacta.sql to change an activity attribute. The latter one, wfchita.sql addresses item attributes.

wfchlut.sql

wfchluc.sql

These two scripts can be used to make changes to your lookups. If you need to change the internal name of the lookup type or of any of the lookup codes use one of these two scripts. Since Oracle Workflow joins tables, in most cases, based upon the varchar values, changing the internal name of an object impacts foreign key relationships. These scripts will make the change requested not jusrt in the lookup table, but in all the places where the old value might appear as a result, too.

Remember, the data cleanup scripts are very useful, but they are so useful because they're so powerful. Make sure that you know what a script does before you run it and test, test, test.

That was just a sampling of some of the scripts that Oracle delivers with a Workflow installation. Of course, I don't recommend that you try to memorize these, but it is important that you know of their existence and where to find them when needed. You can obtain a complete listing of all of the scripts in Chapter 14 of the Oracle Workflow Guide[1]. You also can find more information on Metalink[2]. As an alternative, you can copy the scripts themselves right from your server (just look in the directory referenced above on page 19), and read the comments and review the actual statements being executed on the scripts.

Chapter 4: WORKFLOW APIs

Earlier in the paper, we examined seventeen of the most useful tables with in the Workflow application. I explained where much of the pertinent data are stored, and how to join the tables in queries. In this section, I would like to expand the topic even more by explaining some of the APIs provided by Oracle to access, insert, and update records in these tables.

An *application programming interface,* is a program designed to permit a developer to get inside a computer application. Most people just call them by the acronym, API. Oracle provides over a hundred public APIs that you can call from PLSQL programs to create and manage Workflow process instances. Since you cannot issue DML commands against Oracle's tables, these APIs permit you to create and launch processes, handle errors, update attributes, and dozens of other functions. The APIs usually are more efficient even for querying data in these tables because they employ caching, resulting in fewer context switches between the PLSQL and SQL engines.

---

[1] Oracle Workflow Guide, version 2.5. Chapter 14, January 2000. Part Number A75397-01

[2] Metalink Note 108185.1

The Workflow APIs are available as PLSQL procedures and functions or in java methods, but in this paper, we will focus only on the versions which are implemented in PLSQL. You'll find many of them in the wf_engine package, but there are others in wf_purge and wf_notification, just to name two.

For just a moment, recall the dichotomy I used to divide the Workflow tables between those that contain setup information and those with runtime data. There are not any APIs that make changes to the setup tables; updates to an item's definition should be made using the Workflow Builder tool. All of these APIs read and changes to the tables containing runtime information because they are intended to be used so that a running Workflow process can have access to information about itself and make changes to that information as needed. Complete documentation about all of the publicly available APIs can be obtained in the Oracle Workflow API Reference Guide[3], which is available on Metalink.

It bears repeating that if a procedure is called from a Workflow activity node, it should not include any COMMIT statements. So, when calling of these APIs that modify data, such as the APIs that update attribute values, you'll have to trust that Oracle Workflow will commit the transaction once the activity is complete. On the other hand, when these APIs are called from outside of Workflow, a commit will be required.

While there are many APIs that update Workflow information in the database tables, a number of them do nothing more than retrieve data. That fact brings to bear a question: why not just query the tables yourself? There are two reasons that you should rely on the available APIs.

First, the APIs are more efficient. The Workflow Engine caches data in PLSQL collections to avoid excessive context switches caused by repeated calls for process information. For example, Workflow coding frequently involves a succession of calls to read item attributes. By using the APIs, you can leverage this efficiency.

Second, when you use the APIs, you get code that Oracle will maintain for you. If the data model were changed, Oracle would update the APIs to reflect the change. You would be on your own to fix any queries that you wrote.

CREATING AND LAUNCHING PROCESSES

To begin our examination of the Workflow APIs, we'll start in the same place Workflow itself begins, by creating and starting processes. We'll consider the two APIs in tandem, and then look at a second option to accomplish the same functionality.

```
wf_engine.createProcess
(   itemtype    IN VARCHAR2
,   itemkey     IN VARCHAR2
,   process     IN VARCHAR2 DEFAULT ''
,   user_key    IN VARCHAR2 DEFAULT NULL
,   owner_role  IN VARCHAR2 DEFAULT NULL);

wf_engine.startProcess
(   itemtype  IN VARCHAR2
,   itemkey   IN VARCHAR2);
```

The createProcess procedure essentially does nothing more than insert a new record into wf_items. At that point, the newly created item exists as a valid item, complete with a begin date and an item_key, but it has not begun transitioning through the various activities in the process. Even though it has not begun transitioning, the attributes associated with the item are available for updating via the SetItemAttr APIs that we'll see later in this section on page 26. The second API listed above, startProcess, is the one that actually initiates process flow by executing the first activity in the process.

---

[3] Oracle Workflow API Reference, Part Number B10286-02

© Copyright 2008 by Daniel Stober

It's up to you to supply an itemkey for your process. The API will check to make sure that the itemkey value is unique within that itemtype. The itemkey can be any combination of letters, numbers, and underscore characters, as long as it is unique. The methodology for how the itemkey is generated is left up to you. You can use a sequence or some other identifier specific to your process as long as you can ensure that the value is unique.

(As an aside, often, when I am in the development phase and running an item over and over, I often string out the time of day and use that for my itemkey. An itemkey generated with `TO_CHAR(SYSDATE, 'HH24MISS')` is assured to be unique unless I happen to try it at the exact same second on different days, That way, I don't have to be preoccupied with finding unique keys while focused on development and testing. This is not a sound strategy for production, but it works in development.)

The only two parameters that you must include in all cases are the first two, itemtype and itemkey. The other parameters are optional, unless there is no selector function defined for the item_type. (We will cover how to write a selector function in Unit IV, beginning on page 52.) If there is no selector function, then the "process" parameter is required also, so that the Workflow engine knows what process within the item_type should be started. This is true even if there is only one runnable process within the item. Whether the name of the process to be started comes from the "process" parameter or from a selector function, that value is stored in the root activity field.

The other optional parameters are "user_key" and "owner_role" which allow you to specify a user-friendly identifier for the item and the name of role that will be designated as the owner of the item. If you create a process without specifying a value for one or both of these and you later change your mind, you can use the set_item_user_key and set_item_owner APIs found in the wf_item package to set them later.

As an alternative, there is another API which wraps the functionality of the two APIs above by combining them into a single procedure:

```
wf_engine.launchProcess
(   itemtype    IN VARCHAR2
,   itemkey     IN VARCHAR2
,   process     IN VARCHAR2 DEFAULT ''
,   user_key    IN VARCHAR2 DEFAULT NULL
,   owner_role  IN VARCHAR2 DEFAULT NULL);
```

Use this API if there is no need to address any of the attributes before the process instance is started. It will both create the item, which is explained above, and begin executing the activities of the process. LaunchProcess has the same three optional parameters available that are found in createProcess, and, like createProcess, "process" is required if the item_type has no selector function.

This following anonymous block illustrates how easy it is to use this statement to create and start a Workflow process instance. Notice that this block includes a commit statement; the example assumes that this procedure was not called from within another Workflow process.

```
BEGIN
   wf_engine.launchProcess
        (  itemtype    => 'DJS_DEMO'
        ,  itemkey     => '123'       );
   COMMIT;
END;
```

ITEM ATTRIBUTE VALUES APIs

Next, we'll look at some APIs that address item attribute values. The beauty of Workflow comes from its simplicity. For the most part, the entire logic of a running process is based upon values in results and attributes and the interplay between them. Once you understand how to manipulate attributes in a Workflow, you can accomplish virtually anything. Thus, understanding how to use these APIs is key because it's likely that most of the programs you'll write will rely heavily on addressing attribute values.

We'll start with the APIs implemented as PLSQL functions that read attribute values and return those values to you. There are three different basic varieties of this API; choose the one of the three that matches the underlying datatype of the attribute being queried.

For attributes that have a datatype of "Number" and "Date" use getItemAttrNumber and getItemAttrDate respectively, and those functions will return a value that is either a PLSQL NUMBER or DATE. For attributes with a datatype of "Text", use getItemAttrText and the function will return a VARCHAR2 value. While it seems unlikely that you would not know that data type in advance since you are the developer, if there is any uncertainty about the datatype, you can use the getItemAttrInfo API to find out first, or you can always use getItemAttrText, as it is generic.

All three of them require that you pass in an item_type, item_key, and the name of the attribute from which you are seeking the value. The name that should be supplied is the internal name of the attribute – and, don't forget that since this is a text string comparison, it is case sensitive, so the value for aname should be entirely in capital letters.

```
wf_engine.getItemAttrText
(   itemtype  IN VARCHAR2
,   itemkey   IN VARCHAR2
,   aname     IN VARCHAR2)
   RETURN VARCHAR2;

wf_engine.getItemAttrNumber
(   itemtype  IN VARCHAR2
,   itemkey   IN VARCHAR2
,   aname     IN VARCHAR2)
   RETURN NUMBER;

wf_engine.getItemAttrDate
(   itemtype  IN VARCHAR2
,   itemkey   IN VARCHAR2
,   aname     IN VARCHAR2)
   RETURN DATE;
```

Taken together, these are three of the most common and useful APIs. Using the three APIs above, in combination with the next set of APIs, which permit you to set the values of attributes, permits you to write PLSQL code that can interact with the Workflow process that called it.

> GOTCHA: Use the Internal Name of the attribute – not the Display Name – entirely in capital letters.
>
> In specifying the attribute name in these and other APIs, the value you supply for the "aname" parameter should be the internal name of the parameter. Also, remember that since the test for the name is a string comparison, it is case sensitive. Internal names always should be typed entirely in capital letters.

If you try to call this API against an item_key, item_type, attribute combination that does not exist as a record in wf_item_attribute_values, you can expect this error:

```
ORA-20002: 3103: Attribute 'MY_ATTR' does not exist for item 'DJS_DEMO/123'.
```

If you have kept up with patching, each of these three APIs now includes an optional fourth parameter which accepts a Boolean value and tells the function to ignore errors that would otherwise be raised if the attribute specified for “aname” does not exist. This functionality could be useful in situations where the API might be called against processes running that were launched under differing versions. If this behavior is desired, specify TRUE for the parameter “ignore_notfound”; otherwise just disregard it.

Be aware that you will see the same exception raised whether the call contains an invalid attribute name, item_key, or item_type. Similarly, the ignore_notfound parameter will suppress exceptions raised in any of these conditions.

No exceptions will be raised if you call the wrong variation of the API for the datatype of the attribute. For example, if the attribute is defined to be a “Date”, and you call the getItemAttrNumber, it will return NULL, since the value in “number_value” field in wf_item_attribute_values for this record will be null. It bears repeating to point out that you, as the developer, should be careful to insure that you use the correct version of the API.

In conjunction with the three getItemAttr APIs are the following three which allow you to set the value of an item attribute. Of course, you have to know the datatype of the attribute being updated before you can know which one of these APIs to use.

```
wf_engine.setItemAttrText
(   itemtype  IN VARCHAR2
,   itemkey   IN VARCHAR2
,   aname     IN VARCHAR2
,   avalue    IN VARCHAR2);

wf_engine.setItemAttrNumber
(   itemtype  IN VARCHAR2
,   itemkey   IN VARCHAR2
,   aname     IN VARCHAR2
,   avalue    IN NUMBER);

wf_engine.setItemAttrDate
(   itemtype  IN VARCHAR2
,   itemkey   IN VARCHAR2
,   aname     IN VARCHAR2
,   avalue    IN DATE);
```

The parameters in the call signatures are identical to the getItemAttr APIs, with two differences: first the presence of a fourth parameter, “avalue”, for which you supply the actual value to which the item_attribute should be updated, and second, there is no return statement because these APIs are PLSQL procedures – not functions.

Unlike the getItemAttr APIs, there is no parameter to ignore exceptions. If there is doubt as to whether the attribute exists, you’ll have to test for it in your code before attempting to set the attribute value or you’ll have to handle the exception when raised.

ACTIVITY ATTRIBUTE VALUES APIs

The activity attribute APIs are similar to the item attribute APIs as far as the “get” functionality goes. But, it’s important to note that there are no “set” APIs for activity attributes. Activity attributes derive their values either from the value of an item attribute or from a constant established at the time of development. That’s the reason that I classified wf_activity_attr_values as one of the definition tables Unit I. Activity attributes, per se, cannot be updated by a process at runtime.

For the "get" functionality, there are three flavors for the API, just as there are for the item attribute APIs. They look like this:

```
wf_engine.getActivityAttrText
(    itemtype IN VARCHAR2
,    itemkey  IN VARCHAR2
,    actid    IN NUMBER
,    aname    IN VARCHAR2 )
    RETURN VARCHAR2;

wf_engine.getActivityAttrNumber
(    itemtype IN VARCHAR2
,    itemkey  IN VARCHAR2
,    actid    IN NUMBER
,    aname    IN VARCHAR2 )
    RETURN NUMBER;

wf_engine.getActivityAttrDate
(    itemtype IN VARCHAR2
,    itemkey  IN VARCHAR2
,    actid    IN NUMBER
,    aname    IN VARCHAR2 )
    RETURN DATE;
```

While it's true that these APIs are similar to the item attribute APIs, notice that each of these three APIs includes a parameter that is not a part of those: actid. Since activity attributes are specific to one activity in a process, that parameter ties the API call to the record in wf_process_activities that identifies the individual activity.

This is how Oracle Workflow can permit you to use the same activity in the same process more than once and still keep track of different values for the same attribute. For example, if you have created a process that employs the "Loop" activity from the "Standard" item type, you may have noticed that you can include more than one loop in a process, and have each of them repeat a different number of times. Oracle accomplishes this by tying the activity attribute to the specific record that represents the activity node in the process.

ITEM ATTRIBUTE VALUES ARRAY APIs

In addition to the APIs I documented here that get and update attribute one at a time there are six more APIs that allow you to read and address item attribute values in bulk. Often, Workflow coding turns into a series of calls to address attribute values, so the array APIs allow you to reduce context switches by gathering and updating many values at the same time. Like the individual API calls, you still have to group your array calls so that your collection consists of attributes of the same data type.

Attribute Array APIs in the wf_Engine package	
SetItemAttrTextArray	getItemAttrTextArray
SetItemAttrNumberArray	getItemAttrNumberArray
setItemAttrDateArray	getItemAttrDateArray

Figure 14

ITEM STATUS APIs

The next two APIs allow you to find out what's going on in your process. Although they have similar names, they answer different questions.

```
wf_engine.ItemStatus
(    itemtype IN  VARCHAR2
,    itemkey  IN  VARCHAR2
,    status   OUT VARCHAR2
,    result   OUT VARCHAR2 );

wf_engine.ItemInfo
(    itemtype  IN  VARCHAR2
,    itemkey   IN  VARCHAR2
,    status    OUT VARCHAR2
,    result    OUT VARCHAR2
,    actid     OUT NUMBER
,    errname   OUT VARCHAR2
,    errmsg    OUT VARCHAR2
,    errstack  OUT VARCHAR2 );
```

itemStatus and **itemInfo** both are APIs from the wf_engine package and both of them are PLSQL procedures. The difference is that the first one, itemStatus reports on the status of the root process (or the process as a whole) while itemInfo provides information about the status of the current activity or the last activity completed. The effect of this difference is that ItemStatus can return 'ACTIVE/#NULL' for the status of the process even while itemInfo returns 'ERROR/#EXCEPTION'. Again, this happens because one API reports on the status of the process instance and the other on the last activity executed in the process.

So if your program requires information about the status of the process as a whole, use itemStatus. To learn what happened in the last activity, call itemInfo.

PURGE APIs

Next, we'll look at a couple of APIs that can be used to clean up workflow data.

```
wf_purge.items
(   itemtype  IN VARCHAR2 DEFAULT NULL
,   itemkey   IN VARCHAR2 DEFAULT NULL
,   enddate   IN DATE     DEFAULT SYSDATE
,   docommit  IN BOOLEAN  DEFAULT TRUE);


wf_purge.activities
(   itemtype  IN VARCHAR2 DEFAULT NULL
,   name      IN VARCHAR2 DEFAULT NULL
,   enddate   IN DATE     DEFAULT SYSDATE);
```

The first of these two APIs, Items deletes records for process_instances from the various runtime tables, while the second in deletes records from the definition tables. The procedures delete data that have an end_date earlier than the date provided in the parameter "enddate".

Look at the call signatures in both of these procedures and notice that there is a default value for each one of the parameters. In PLSQL, this means that all of the parameters in both of these parameters are optional. If a value is not supplied for a parameter, Oracle uses the value specified as the "default" when identifying the items to be purged.

For example, when calling the items API, you could specify a value only for the item_type, and the procedure would act on all process instances of that item_type, regardless of the item_key or end_date. Or, you could supply an item_type and end_date to purge all process instances in that item_type older than the designated date, regardless of item_key. But, fail to provide a value for any of the parameters, then this API will purge all process instances in all item_types, regardless of item_key or end_date.

One thing to remember, in PLSQL null values never match an actual value, so regardless of the value supplied for the date parameter, processes instances and activities which have not completed will not be purged by these APIs. This represents a significant difference between the APIs in wf_purge and the item clean-up scripts documented in the earlier section.

The fourth parameter allows you to decide whether intermediate commits will be performed as the deletions are executed. Here, the default value is "true," meaning that if you call this procedure without specifying values for any of the parameters, then the API will delete all of the runtime data, and then commit it. If you think this sounds like a couple of the scripts we reviewed earlier, you're right!

The items API is used to delete runtime data, and can help to clean up old process instances. The second of these two, activities, deletes records from the definition tables, but only in cases where the activities are no longer included in any process instance records or in any records in wf_process_activities. This limitation is designed to maintain deta integrity: you must delete the runtime data before you can delete the underlying definition data on which the runtime data data relies.

NOTIFICATIONS APIs

You can use the following two APIs to get information about the content of a notification. Pass in nothing more than the notification_id, represented by "nid" in the call signatures below, and these will return either the subject or the body of the notification, with all of the tokens replaced by their actual runtime values.

```
wf_notification.getSubject
(   nid     IN NUMBER)
   RETURN VARCHAR2;

wf_notification.getBody
(   nid      IN NUMBER
,   disptype IN VARCHAR2 DEFAULT '')
   RETURN VARCHAR2;
```

The getBody API can take an optional second parameter specifying the display type. If you supply "text/html", you'll get the HTML version of the notification. Otherwise, just leave that parameter blank to get the text version.

UNIT III: WRITING PLSQL TO IMPLEMENT WORKFLOW FUNCTION ACTIVITIES

While it's true that Oracle provides dozens of functions for you to use in your Workflow process, sometimes you still may find that you need something little different that what is delivered. When that happens, you can write your own code and call it from within your own Workflow. In this section we'll show you how to add some custom PLSQL to your process definition.

But before we do that, I'd like to begin by defining a couple of terms. We'll start at the top with *PLSQL,* a term which already has been used liberally throughout this paper. PLSQL, which also is written as PL/SQL, is short for Procedural Language / Structured Query Language, Oracle's programming language for use within the database. PLSQL is perfect for database work because it integrates most of the familiar SQL keywords for reading records and writing in the database and encloses them in a structured language that supports looping, branching and other common programming constructs.

Now, we move to what is potentially the most confusing usage of a term. In Workflow, a node that implements a stored procedure is called a *function activity,* or often, just a *function*. Be careful not to confuse this with the PLSQL structure of the same name. In PLSQL parlance, a function is a keyword identifying a particular program structure that returns a single value.

Here's where it gets confusing: While "function" is a keyword in both Workflow and PLSQL, the PLSQL code called by a Workflow function is not a function at all in the PLSQL sense; it is a *procedure*. A procedure is another PLSQL structure, but it differs from a function in that it does not return one single value. A procedure still can pass values back to the caller, but it does so using outbound parameters. If you're not a programmer, this distinction may seem a little esoteric, but it will be important as we explore methods available to pass information from PLSQL back to the calling node in a Workflow.

Just remember, a Workflow *function* calls a PLSQL *procedure.*

PROCESS OVERVIEW

When writing stored procedures to be implemented within a Workflow process, the general strategy you'll use is to read the information that the Workflow engine passed into your procedure, interpret that information and carry out processing, and then pass some sort of information back to the calling Workflow process instance. On the face of it, this may not seem too different from the programming involved in any other circumstance – and the truth is that it probably isn't. But there are still some nuances involved in dealing with Workflow that make this an important topic.

The Workflow Engine supplies four discrete data for each call that it makes. We'll delve into those in more detail after the introduction. Your procedure can get additional information about the process by reading values from item attributes and activity attributes.

In the second phase of the strategy, the processing carried out by a custom PLSQL procedure called from a Workflow process can run the entire gamut of the capabilities of the language. For example, within a PLSQL procedure implementing a function activity node, among other things, you can:

- Launch or interact with another Workflow process
- Request a concurrent program
- Perform DML, inserting, deleting, or updating records in database tables
- Pass information back to the Workflow process that made the call

The activities we'll explore will focus on the last of those tasks, relaying information back to the process instance. Since the code implementing a Workflow activity is a procedure and not a PLSQL function, it does not have a specific function return value. There are still two methods available to your PLSQL procedure to communicate with the calling function activity:

- Passing a value in the resultOut parameter
- Updating itemAttributes

THE CALL SIGNATURE

Before we get deeper into the look at what you can accomplish in a custom procedure, let's begin by looking at what information the Workflow Engine will make available to your procedure. Any PLSQL procedure called by a Workflow process from a function activity node should have the following signature:

```
PROCEDURE xxxx
    (   itemtype         IN  VARCHAR2
    ,   itemkey          IN  VARCHAR2
    ,   actid            IN  NUMBER
    ,   funcmode         IN  VARCHAR2
    ,   resultout        OUT VARCHAR2 )
```

The statement above is a parameter list, or *call signature*, for a PLSQL procedure being called by a Workflow process[4]. There are four values that the Workflow engine will pass into the procedure. Depending upon the purpose of the program and the way it is written, the program may use none, one, some, or all of the parameters to complete its task. In addition, there is one parameter reserved to be used by the PLSQL procedure to pass information back to the Workflow engine.

The call signature for function activities			
Parameter	Direction	Datatype	Purpose
itemtype	IN	VARCHAR2	Item_type of the calling process instance
itemkey	IN	VARCHAR2	Item_key of the calling process instance
actid	IN	NUMBER	Instance_id of the activity from which the call originates
funcmode	IN	VARCHAR2	RUN, CANCEL, or TIMEOUT
resultOut	OUT	VARCHAR2	Activity status and activity result

Figure 15

INBOUND INFORMATION FROM THE CALL SIGNATURE

The first two parameters, itemType and itemKey, together denote the unique instance of the process, as it is identified in the Oracle Workflow runtime tables. These two values alone often are sufficient to perform many tasks, including reading the attribute values of the process and updating them.

The next parameter, actid, is the value of "instance_id" of the activity that called the procedure. It uniquely identifies a single record in the wf_process_activities table because it ties directly to the "instance_id" field in that table. If the same function implements multiple nodes within the Workflow process, this value allows the procedure to determine exactly whence the call originated. Also, if the Workflow process has multiple versions active in the database and it's important to know which version of the process activity is in use for this particular process instance, the actId parameter will make it easier for the developer to write logic which ascertains under which version of the process this particular item_key is running. (If you need to refresh your recollection, you can review the discussion about wf_process_activities on page 11.)

In the AND-JOIN procedure in Workflow's Standard item type, the value for the actid parameter is used to query wf_transitions to determine whether any more transitions from activities are left to be fulfilled.

```
SELECT wa.begin_date
  FROM applsys.wf_process_activities  wpa
  ,    applsys.wf_activities          wa
 WHERE wpa.instance_id = 185130
  and wa.name      = wpa.process_name
  and wa.item_type = wpa.process_item_type
  and wa.version   = wpa.process_version
```

The final incoming parameter is funcmode. In most cases, the Workflow engine will pass in "RUN", signifying that the process is moving forward through the flow. But, at other times, especially in error clean-up modes, this parameter could be "CANCEL" or "TIMEOUT". Using this parameter

[4] Page 7-2 Oracle Workflow Guide version 2.5

 © Copyright 2008 by Daniel Stober

allows you to write a procedure that behaves differently when the procedure is called as a result of unwind logic than when it is called for normal process running.

The first time that a process transitions to an activity node, the Workflow Engine always supplies "RUN" as the value of the funcmode parameter. However, in subsequent calls, the value passed in the funcmode parameter in a function activity call is dependent upon the value of the "On Revisit" property in the activity setup.

The fifth parameter is designated as an outbound parameter, and it allows the PLSQL procedure to send information back to the Workflow engine. The engine expects the resultOut parameter to be in a specific format, so that it will know how to handle it. We'll cover that it greater detail in the next section.

The Workflow Engine handles the details of passing the various parameter values in to and out of the procedure, so all the PLSQL developer has to do is make use of the information in the program she writes. With just this seemingly small collection of information, the programmer can derive everything that is needed.

INFORMATION FROM ATTRIBUTES

In addition to the information that the Workflow Engine passes into your procedure, you probably will need to gather additional information about the process. Your procedure can access that information by reading values from attributes. Use the APIs that I referenced in the prior chapter (page 26) to read them.

Of course, in order to call use those APIs, you will have to use the information passed in the parameters.

RETURNING THE RESULT

The fifth parameter in the standardized call signature is the OUT parameter, resultOut. Although there is only one outbound parameter, the Workflow Engine expects to find two discrete pieces of information, concatenated into a single string. The Workflow Engine then parses the string to derive those two values. The two pieces of information that should be included are the status and result of the activity. You don't have to assign any value into this parameter, but if you do, it must be in a prescribed format.

The value assigned to the resultOut parameter should be a single string concatenating the value for actvity_status with the result_value and a colon between them. Properly formatted, the value for a result should look like this:

```
PROCESS_STATUS:RESULT_VALUE
```

Remember, the colon must be included.

We will first examine the two components of the result, and then I'll cover the algorithm that the Workflow Engine uses to parse it and supply defaults if one or both parts are missing.

PROCESS STATUS

The first half of the resultOut parameter, the portion which precedes the colon, should contain the process status. Process status is how you, as the PLSQL developer, can relay information about the success of your procedure from the PLSQL execution to the Workflow process that called it.

When assigning a value for process status, specify one of the values recognized by the Workflow Engine as a valid process status. There are seven different possibilities and those values are declared as constants at the top of the package specification of the wf_engine package.

Value	Constant	Meaning
COMPLETE	`eng_completed`	Activity was completed normally
ACTIVE	`eng_active`	Activity is still running
WAITING	`eng_waiting`	Activity is waiting to be run
NOTIFIED	`eng_notified`	Notification has been sent; still open
SUSPEND	`eng_suspended`	Activity was suspended
DEFERRED	`eng_deferred`	Activity was deferred
ERROR	`eng_error`	Activity was completed, but with an error

Figure 16

In reality, you'll probably never use more than a couple of these in custom code, but I've included all of the values for completeness. In nearly all instances, the code you write would likely return either "COMPLETE" or "ERROR". By the way, if you would prefer not to hard-code values, you can use the constants to supply process_status values. If you decide to reference one of these constants in your code, the parameter assignment might look like this:

```
resultout := wf_engine.eng_completed;
```

The value for process status that is passed back is read by the Workflow Engine, and then stored in the wf_item_activity_statuses table in the "activity_status" field.

ACTIVITY RESULT

The second half of the resultOut parameter, the portion of the string following the colon, should contain the result that the procedure is passing back to the Workflow process. The Workflow engine will store this result in the "activity_result_code" field in wf_item_activity_statuses. The activity result is important for transitions. If the process contains logic that relies on a result value, then that result code is crucial.

Ideally, the value passed back should map to a valid lookup_code record in wf_lookups for the lookup_type defined as the result type for the activity. If the activity returns no result, the value will be "#NULL", which is the constant eng_null from the wf_engine package.

For example, if you are writing PLSQL to implement an activity node that has "WFSTD_COMPARISON" defined as the result_type, you would want to ensure that the result passed back to the node is one of the lookup codes that make up the "WFSTD_COMPARISON" lookup type. Perhaps, the resultOut might look like this:

```
resultout := wf_engine.eng.completed || ':' || 'EQ';
```

The result code value is not validated by the Workflow Engine, so any string following the colon in resultOut will be saved in the "activity_result_code" field, even if it does not match any of the lookup codes. If you supply a result_code containing more than 30 characters, only the leftmost thirty will be saved because that is the defined length of the target field – the Workflow Engine will not raise an exception.

Since the result_code is not validated against the lookups, it is important that you, as the developer, ensure that the lookup_code your PLSQL procedure supplies matches one of the lookup_codes. The reason that this is key is that records in wf_activity_transitions rely on valid result_codes to drive process flow. If the Workflow engine cannot find a transition record to match your result code, the process will become #STUCK.

The other values that can be placed into activity_result are "DEFERRED", "NOTIFIED", and "WAITING" but you will not need them for any of the coding that will be covered in this paper. You can find a complete list on Page 7-4 of the Workflow Developers Guide[5].

PARSING THE RESULTOUT

If you choose not to provide a resultOut, the Workflow Engine will supply "COMPLETE" for the activity status and "#NULL" for the activity result in the item_activity_statuses table. Alternatively, you also may choose to supply only one of the two components of the resultOut. If you supply only a partial result, the Workflow Engine will attempt to determine which of the two halves you provided and then will supply the default for the other half.

In parsing the resultOut, the Workflow Engine applies the following three step test:

1) Does the resultOut string contain a colon? If so, then the activity status is to the left of the colon, and the activity result is on the right.
2) If no colon, does the value in the resultOut string match any of the expected values for activity status ("COMPLETE", "NOTIFIED", "ERROR", etc)? If so, save the value of the status, and record "#NULL" for the activity result.
3) If unable to match with a known status value, then assume the value in the string is an activity result. Save "COMPLETE" to the status field, and record the value from the string as the activity result.

Figure 17 shows some examples of the parse algorithm at work:

resultOut value		Activity_status	Activity_result
COMPLETE:Y	=>	COMPLETE	Y
wf_engine.eng_completed:Y	=>	COMPLETE	Y
wf_engine.eng_completed	=>	COMPLETE	#NULL
Y	=>	COMPLETE	Y

[5] Page 7-4. Verison 2.5 of the Guide.

wf_engine.eng_error	=>	ERROR	#NULL

Figure 17

How the Workflow Engine parses various combinations of activity_status and activity_result.

PARAMETER NAME SPELLING

A common question asked by people who are new to writing code for Oracle Workflow is, "Do I have to spell the names of the parameters like that?" After all, some shops have coding standards that discourage abbreviations when not necessary or that specify a specific prefix to identify parameters. Others just wonder why "item_key" is spelled with an underbar when used as a field in the table, but is concatenated to a single word when used as a parameter. It's a fair question.

The answer that I have given in presentations over the past few years is "No, but I highly recommend it." I generally advise following the instructions in the documentation. Just because some practice which is contrary to the documentation works today is no guarantee that it will continue to work that way in the future.

In my coding, I spell the parameters exactly as they are spelled here. When it comes to item type and item key, I leverage the fact that the parameter name is spelled different from the field name to my advantage. I have learned to use that as a visual cue to distinguish between the two when I am reading my code.

With that having been said, I have softened my position. As I have read more of the delivered code, I have noted that Oracle's own developers don't always adhere to the prescribed parameters. They frequently use "funmode" instead of "funcmode". Some groups have strayed so far as to prefix the parameter names like this: "p_itemtype".

The reason it works is because the Workflow Engine executes the procedure for a function activity by building and executing a dynamic PLSQL statement within the wf_engine_util package. That dynamic PLSQL creates a call that passes parameters by position, which means that the names of the parameters never come into play.

Of course, Oracle could decide to revamp the wf_engine_util package and invalidate what I wrote above, but I personally believe that it is unlikely, given that (1) it would break their code, also, and (2) Oracle Workflow is a product which is in the later stages of its life cycle. At this point, I say that best practices dictate that you stick to the documentation, but I have laid out the facts so you can make your own decision.

NO COMMITMENT

One the cardinal rules of writing a custom procedure to implement a Workflow function activity is that the procedure cannot issue a COMMIT. For long time database developers, omitting a COMMIT statement from a procedure which is calling APIs and updating table values seems almost unnatural, but you just have to trust that Workflow will handle it

Oracle Workflow handles transaction management itself behind the scenes by creating a SAVEPOINT before each activity node as it transitions through a process. If an error occurs in the activity, the Workflow Engine rolls back the transaction to that savepoint.

When a PLSQL call is made to implement a function activity, that call occurs within the same database transaction as the Workflow engine's savepoint. So, if that code issues a COMMIT, the entire transaction is committed, effectively erasing the savepoint, and

meaning that Workflow will be unable to roll the transaction back to the state as it existed before the activity began if it encounters a subsequent error.

> GOTCHA: DDL statements result in an implicit commit.
>
> If your procedure includes a DDL command like "TRUNCATE TABLE" you have just issued a commit – even though you didn't type the word! Thus, DDL statements should not be included in procedures called by Workflow.

Similarly, you should not place an unqualified ROLLBACK into your code. If you have trickier logic within your procedure that may require a rollback you can insert a named SAVEPOINT within your procedure, and rollback to your own savepoint in the event of an error. But remember, roll back to a named savepoint only.

If neither of these strategies will work for you, and your procedure creates a transaction which must be committed, then you'll have to handle that as a part of an autonomous transaction. When an autonomous transaction is created, it runs in its own session, apart from the session of the Workflow's transaction. Activity within the autonomous transaction can be committed and rolled back independently of the main transaction. In fact, since it operates as a separate session, the transaction *must* be committed, because it will not be handled by the Workflow's commits.

To create an autonomous transaction, place the PRAGMA keyword in the declaration of your procedure:

```
PRAGMA AUTONOMOUS_TRANSACTION;
```

However, autonomous transactions should be used very sparingly. Give careful consideration to the question of whether an autonomous transaction is truly needed. Remember, when you include an autonomous transaction, you are sacrificing Workflow's built-in rollback.

NO VERSION CONTROL

While Oracle Workflow supports versioned activities and processes, remember that the PLSQL code that you write is not versioned. When you issue the "CREATE OR REPLACE …" command to update a package or procedure, the code is changed permanently. Workflow process instances that were launched in a prior version will call the new version of the code if they include an activity that calls your code.

For example, if you modify a process to add a new item attribute to a item type and then modify a PLSQL procedure called by an activity in that item type so that it will read or update the value in that attribute. Any process instances that already were running under the prior version would still call the new code when they reached that activity node. If the developer had not taken that possibility into account, those PLSQL calls likely would result in an error.

So, how can the developer avoid such a problem? There are two different approaches.

The first way to handle this would be to create a new procedure with a different name. This technique would mean that processes running under the old version would still call the old procedure, but newly launched processes would call the new procedure. Of course, one disadvantage to this approach is that you would have to maintain some kind of tickler to remind you to remove the old code later, once all of the old processes had completed.

© Copyright 2008 by Daniel Stober

The second approach is more flexible but it requires a better understanding of the workings of Workflow to be able to finesse it. Essentially, you write logic in your code that allows the code itself to determine whether it was called by an "old version" or "new version" instance of the process. Exactly how you accomplish this would depend upon the circumstances of the process and the changes made, but it usually would involve referencing the "actid" parameter and querying wf_process_actvities. At that point, you could join to wf_activities to determine the version number or the active dates of the process. Be very careful about relying on the version number alone unless you can ensure that the versions in your development environment are in synch with production. Since the development cycle often involves building a process multiple times, the development version number often gets ahead of the production version.

Chapter 6: ERROR HANDLING

Part 1: Workflow Error Process

An important consideration as you get ready to write your own is error handling. Of course, no one wants their program to end in an error, but it happens to the best of us. In order to make sure that your proce make sure that your procedure follows Workflow's defined error handling structure, you have to understand how it works. There two components in exception handler. In this section, we'll cover the error process, and later, exception handler.

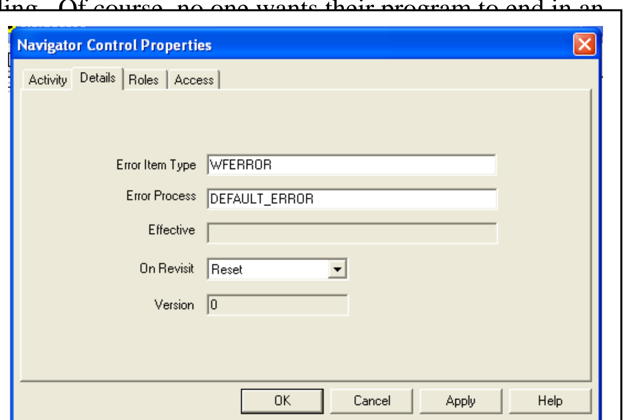

Figure 18

Defining an error process for an activity. In this case, if an error occurs in this activity, the Workflow Engine will launch an instance of the "DEFAULT_ERROR" process in the "WFERROR" itemtype.

In a nutshell, when the Workflow Engine encounters an error condition in a process, it launches another process in response. This is what's called the "error process." The usual purpose of the error process is to alert an administrator about the error so that that person can intervene and resolve the problem.

The process that will be launched is determined during design time. Within the Workflow Builder, you can specify an error process at the activity level and/or at the process level. When an error condition occurs, the Workflow Engine begins looking for a process to launch in response. It starts searching at the activity in which the error occurs. If an error process has been defined for that activity, the Workflow Engine launches an instance of that process. If not, then it checks the process of which the error is a part, and it continues up the chain until it locates an error process.

In the illustration in Fig 19, if an error occurs in this activity as it executes in a process, the Workflow Engine will create an instance of the "Default Error Process" from the "System:Error" itemtype and launch it. In the properties window, the designer has supplied the internal names of that process and itemtype, as is required.

For most purposes, the Default Error Process is sufficient. It serves its purpose of notifying the administrator about the problem, and ample, if the administrator fails to respo error condition still exists before re-not but notify a different administrator, tha

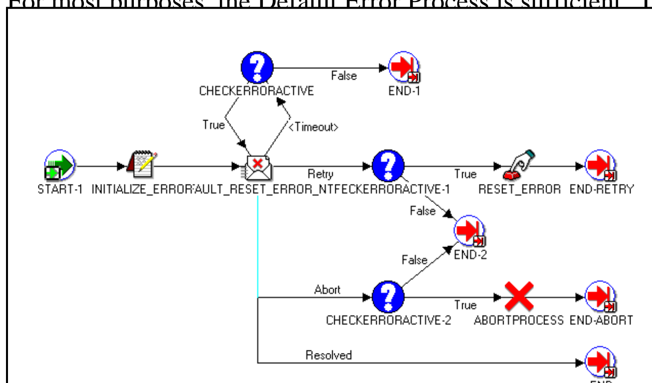

Figure 19

The Default Error Process

Five attributes:

- Error item_type
- Error_item_key
- Error activity_id
- Error message
- Error stack

Figure 20

Five of the twenty attributes for which the WF Engine supplies values to any process serving as an error handler.

ight 2008 by Daniel Stober

Whether you choose to use the Default Error Process or create your own, whenever the Workflow Engine launches any process as an error handler process, it provides values for *twenty* different attributes. If those attributes already exist in the error process, as they do in the Default Error Process, they are updated with information about the current error condition. If any of them do not exist, the Engine will first create them, and then supply the values. The information in these attributes allows the Default Error Process – or your custom process – to include the pertinent information in the notification, so that it can be corrected.

By default, the Default Error Process will notify the role called "SYSADMIN". But, if you would like to use that process but notify a different role, you can do that, too! As an example, say that you have one person who is tasked with handling errors that arise in Fixed Assets Workflow processes, and another who takes care of errors from HR Workflows. You can accomplish this without any modification to the Default Error Process.

The secret is that you need to create an attribute in the process where the error would be raised. This is important; you don't add the attribute to the System:Error item type, it is placed into the itemtype where the error would occur. The attribute that should be added should have an internal name of "#WF_ADMINISTRATOR". Don't forget the pound sign in the first character position. Populate the attribute with the name of the role who should handle errors arising from this item type, and they'll be notified.

Part 2: Error handling in PLSQL

So far, we've seen what Workflow does when an activity results in an error. But how does that happen? We know that, behind the scenes, an activity is implemented by a PLSQL procedure. If the error happens in the realm of the PLSQL, how does information about the location and cause of the error get back to Workflow? In this section, we'll answer that question.

Workflow PLSQL Error Handling:
1. Trap the exception
2. Record the Details
3. Re-Raise the Exception

The basic model that you should be followed is that an exception should be trapped in PLSQL, the details about the error should be recorded, and then the error should be re-raised. We'll talk in more detail about each of these in the coming paragraphs.

TRAP THE EXCEPTION

PLSQL, like virtually all programming languages, provides a mechanism for a developer to handle situations that arise in a program that will cause the intended purpose of the program to fail. In broad terms, this scheme is known as "error handling". The errors that arise might be situations that the developer anticipated or runtime errors that were not expected. In any case, a developer has to expect that bad things will happen to his program, and write routines that permit to program to behave gracefully in those situations. In workflow, the graceful behavior is accomplished by means of the error handling scheme that I described in the previous section.

In order for Workflow to handle the "graceful" part, the developer must first do her part. The first of these is remembering to include an exception section in every procedure written to implement a function activity. A full discussion of PLSQL error handling is beyond the scope of this paper, but suffice to say that PLSQL provides a structure for a developer to include statements that will be carried out if any error occurs during program execution. Collectively, those statements are called an "**exception handler**".

Your procedure can contain as many exception handler sections as necessary for you to implement the logic you are modeling. But, if you want to ensure that the exception you are handling results in the launching of the error process within Workflow, you must take care to propagate the exception out of your procedure. By "propagate" I mean, make sure that your procedure does not handle the condition completely; you want the PLSQL Engine to continue looking for an error handler. Generally, this is accomplished by a "RAISE" statement within the error handler, but you can also set the resultOut to "ERROR" (or "wf_engine.eng_error" if you prefer to use the constants).

The way that this usually looks within the confines of Workflow programming, is a "WHEN OTHERS THEN" exception section at the bottom of the procedure. Any error raised will cause program execution to jump to this section. Once you handle the logging details that we'll discuss in the next section, you include the "RAISE" statement. Essentially, this instructs the PLSQL engine to create the same error condition over again. Your program is saying essentially, "I know I just told you that I wanted to take care of errors when I included this exception handler, but I've changed my mind. Let the calling program deal with it instead."

Of course, in PLSQL, exception handler sections are optional. If you don't include the exception handler section, the default behavior is to propagate the error back to the caller. So, why not just omit the exception handler section altogether? The reason is that we need to log information about the error condition to the error stack so that the Workflow error process will have access to it. We'll see how that works momentarily. For now, just know that you have to handle the error and then re-raise it.

RECORD THE DETAILS

This is the most important part of the error handling steps. Once your code has trapped an error that needs to be propagated upward, you first need to add the details onto the error stack. This is accomplished by use of a public API in the wf_core package.

```
wf_core.context(pkg_name  IN VARCHAR2
       ,  proc_name       IN VARCHAR2
       ,  arg1  IN VARCHAR2 DEFAULT '*none*'
       ,  arg2  IN VARCHAR2 DEFAULT '*none*'
       ,  arg3  IN VARCHAR2 DEFAULT '*none*'
       ,  arg4  IN VARCHAR2 DEFAULT '*none*'
       ,  arg5  IN VARCHAR2 DEFAULT '*none*'
       ,  arg6  IN VARCHAR2 DEFAULT '*none*'
       ,  arg7  IN VARCHAR2 DEFAULT '*none*'
       ,  arg8  IN VARCHAR2 DEFAULT '*none*'
       ,  arg9  IN VARCHAR2 DEFAULT '*none*'
       ,  arg10 IN VARCHAR2 DEFAULT '*none*');
```

Figure 21

The complete signature of the wf_core.context core API. Use this to place on the error stack as much information as is necessary about any error arising in your workflow activity.

Within the package, you should call the procedure called "context," which allows you to pass in the name of the package and procedure where the error was raised, along with ten additional pieces of information of your own choosing. By convention, the first four of those values are used for the four inbound parameters in the procedure, although this is not a requirement. The fifth optional argument is generally used to record the actual SQL error message. All of these values, and any others you choose to include, are placed on the error stack to allow debugging, and they are stored in the "error_stack" field in wf_item_activity_statuses.

The complete parameter list for wf_core.context is pictured in Figure 21. the first two arguments are reserved for the name of the package and procedure in which the error handler resides. If the procedure is not in a package, then just pass the name of the procedure in place of the first parameter, and then supply "NULL" for the proc_name parameter.

Again, the developer is free to provide whatever information he or she believes will be relevant for troubleshooting, but I would recommend that you make it a best practice to include at least the following:

Recommendations for using wf_core.context	
Parameter	Value
arg1	itemtype
arg2	Itemkey
arg3	TO_CHAR(actid)
arg4	Funcmode
arg5	SQLERRM

Figure 22

So how does all of this look when we put it all together? A procedure implementing a workflow function activity should always include this line in the error handler:

```
EXCEPTION
   WHEN OTHERS THEN
      wf_core.context ('my_wf_pkg', 'my_wf_proc', itemtype, itemkey, TO_CHAR(actid), funcmode, SQLERRM);
```

RE-RAISE THE EXCEPTION

The final piece of the error handling puzzle is need to re-raise the exception after you've recorded the details with wf_core. To do that, use the PLSQL keyword "RAISE". That statement says, "Take the last error and treat it as if it never had been handled." The PLSQL engine has just located an error handler, but you are telling it to find another one. Ultimately, this means that the error will make back to the Workflow Engine and the Workflow Engine itself will have to handle the error. The Workflow Engine will launch the error process associated with the activity, as described in the prior section.

Although using the RAISE statement is the most common approach to propagate the error out, there is an alternative. You can set the status in the resultOut parameter to the value of the constant wf_engine.eng_error, which is "ERROR". (Refer back to the chart in Fig 11 on page 16 or Fig 16 on page 34). From the perspective of the Workflow Engine, this status result has the same outcome as an actual error. The Workflow Engine does not care about the nature of the error and doesn't recognize a difference – all of the information about the error was placed on the error stack by the call to wf_core.context.

One possible advantage of using resultOut to propagate the exception, is that it would allow you to set a result code, too. Of course, since the process would be in error, the process flow would not follow a transition based upon the result code, but if you saw a need to do this, you could.

```
EXCEPTION
   WHEN OTHERS THEN
      wf_core.context ('my_wf_pkg', 'my_wf_proc', itemtype, itemkey, TO_CHAR(actid), funcmode, SQLERRM);
      RAISE;
```

Chapter 7: THREE EXAMPLES

So, to close this unit, I would like to provide some code snippets illustrating processes that incorporate several of aspects of the concept I described and integrates the use of some of Oracle's Workflow APIs.

Over the next dozen or pages, I present to you three examples of code written to implement function activities. I have annotated the code to aid in the explanation. If you take the time to read each of the procedures and make sure that you understand each step, you will derive the maximum benefit from this paper. The three examples I have included are one that modifies an Oracle Workflow process, and two custom Workflow processes.

- Customizing GLBATCH
- Repeat Result
- Set Wait Date

As you read each procedure, watch for how each one uses the inbound parameters differently to fit each individual situation. Also, notice the resultOut works in each one to convey the necessary information back to the Workflow Engine. The third one, “Set Wait Date”, accomplishes its mission by means of updating an attribute.

© Copyright 2008 by Daniel Stober

CUSTOMIZING GLBATCH

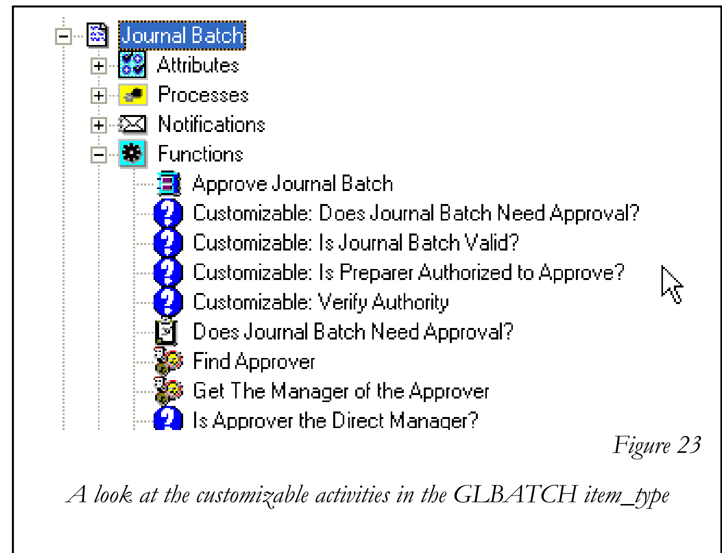

Figure 23

A look at the customizable activities in the GLBATCH item_type

For the first example, we will to turn to the Journal Batch (GLBATCH) item_type from within the E-Business Suite. Inside this item_type are four generic approval activities that permit customers to insert any custom approval conditions necessary to support their own business logic. The four function activities allow you to write PLSQL procedures to implement your custom approval logic and replace the delivered package with the code you wrote.

Each of the four function activities have been setup with a Result Type of “Yes/No”, and points to one of four procedures in a package called “gl_wf_customization_pkg”. As delivered, each of the procedures in the package is a shell that performs no actual processing and returns a result of “Y”. Oracle expects that customers will insert their code logic directly into one of the four procedures, and then replace the delivered package body. Oracle will not deliver a patch that will stomp over your custom code.

The four procedures in the package are “is_je_valid”, “does_je_need_approval”, “can_preparer_approve”, and “verify_authority.” Those procedures implement the four function activities in the item_type whose display_name begins with the word “customizable”. Determining which procedure implements which function activity should be fairly evident from the names alone, but you can view the properties of each activity and note the value in the Function Name field to be sure.

When a need arises to implement custom logic insert into the process, all you have to do is write your code and then insert it directly into the relevant procedure and then the entire package body. For a customization of the Workflow, you don’t have to modify the Workflow process at all. By the way, remember, if you are customizing only one of the procedures, you still must include the shell code for the other three procedures into the replacement package.

Say, for example, that you needed to add some custom logic into the “is journal batch valid” function activity, which is included as a node in the GL Initialization and Journal Validation Process. A screen shot of that process appears below in Figure 24.

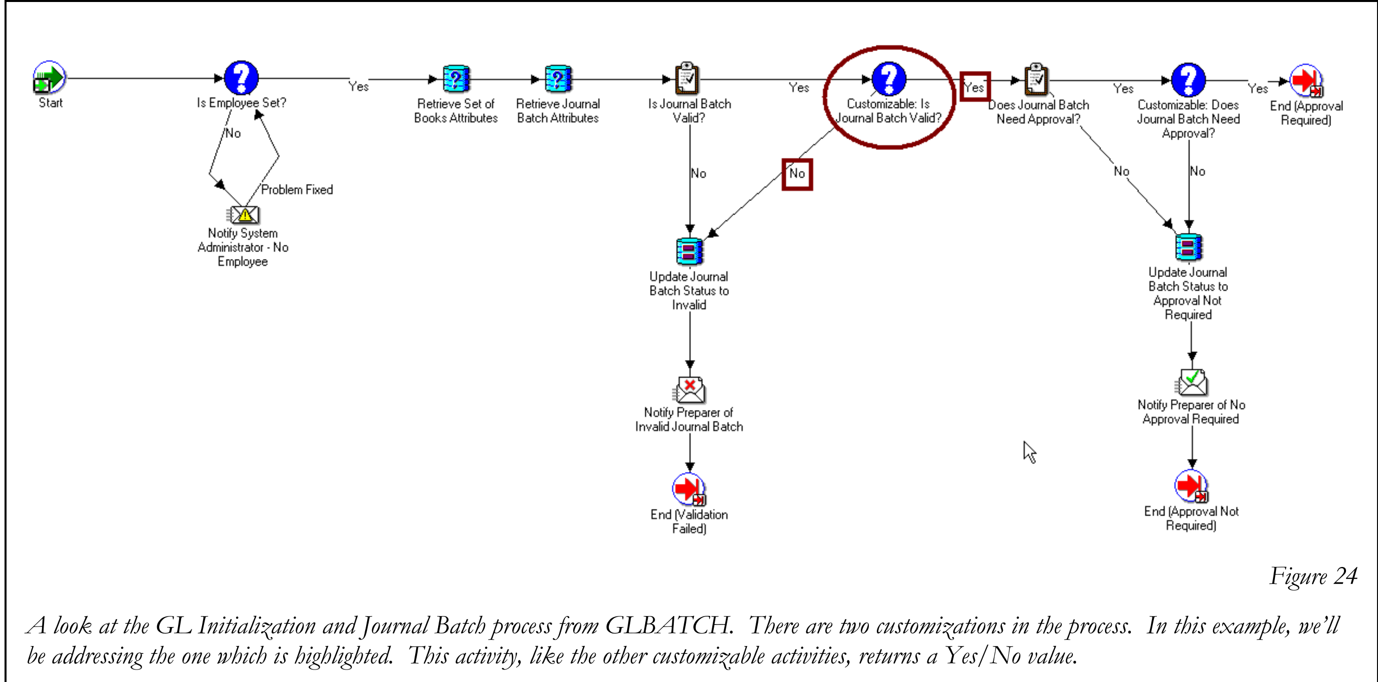

Figure 24

A look at the GL Initialization and Journal Batch process from GLBATCH. There are two customizations in the process. In this example, we'll be addressing the one which is highlighted. This activity, like the other customizable activities, returns a Yes/No value.

In this process, the customizable activity is the one that has been highlighted with a circle, five nodes to the right of the start activity. That activity, "Customizable: Is Journal Batch Valid?", has a return type of Yes/No, just like the other customizable activities in this item_type. Of the two transitions away from that activity node, "No" ultimately leads to a process result of "Validation Failed" while "Yes" allows the process to continue to the approvals evaluation phase.

Notice that the node just prior to the circled one is the delivered "Is Journal Batch Valid" activity. That arrangement essentially permits two different validity checks in the process. First, execute the delivered functionality, and, if the batch passes that test, then apply the custom test. As I mentioned above, by default the code implementing the customizable activity is a shell that always returns "Yes," so if you choose not to insert any custom logic, the process will function just fine.

Before I add customization, I'll show you a snippet of what the procedure looks like as delivered:

```
 1 PACKAGE BODY APPS.GL_WF_CUSTOMIZATION_PKG AS
 2
 3 PROCEDURE is_je_valid(itemtype     IN VARCHAR2,
 4                       itemkey      IN VARCHAR2,
 5                       actid    IN NUMBER,
 6                       funcmode IN VARCHAR2,
 7                       result   OUT NOCOPY VARCHAR2 ) IS
 8 BEGIN
 9   IF ( funcmode = 'RUN'  ) THEN
10     -- Additional code can be added here.
11     -- COMPLETE:Y (Workflow transition branch "Yes") indicates that the journal
12     --            batch is valid.
13     -- COMPLETE:N (Workflow transition branch "No") indicates that the journal
14     --            batch is not valid.
15     result := 'COMPLETE:Y';
16   ELSIF (funcmode = 'CANCEL') THEN
17     NULL;
18   END IF;
19 END is_je_valid;
20
21
22 PROCEDURE does_je_need_approval  . . .
```

```
23
24 PROCEDURE can_preparer_approve . . .
25
26 PROCEDURE verify_authority . . .
27
28 END GL_WF_CUSTOMIZATION_PKG;
```

In the listing above, the delivered is_je_valid procedure is found between 3 and 19. The new logic that we'll write will be inserted in place of those lines. Remember, this procedure is in a package, and there are three other procedures there. When we make changes to the one procedure, we'll have to be sure that we include the other three. If we don't, then the other customizable activities in the item_type that rely on those shells will not function. (Actually, since we are changing only the package body, the PLSQL compiler would reject our changes until we did include those other three procedures.)

So, if you want to modify only one of the procedures, the first step for you will be to get a copy of the delivered package body. You can spool it directly out of the database or take the package creation script off of your database server. To spool it, use this statement:

```
SELECT text
 FROM dba_source
WHERE owner = 'APPS'
  AND name  = 'GL_WF_CUSTOMIZATION_PKG'
  AND type  = 'PACKAGE BODY'
ORDER BY line
```

If you would prefer to use the script from your server instead of spooling it out of the database, then download and open the file, **glwfcusb.pls**.

To keep the example simple, I've assumed that I have a custom function, that accepts a batch_id as a parameter, applies my custom validation checks, and returns true or false. That means that in the customization here, all I have to do is lookup the batch number from an item_attribute, call that custom function, and then translate the function's return into a result acceptable to this activity node.

```
 3 PROCEDURE is_je_valid(itemtype     IN VARCHAR2,
 4                   itemkey         IN VARCHAR2,
 5                   actid           IN NUMBER,
 6                   funcmode        IN VARCHAR2,
 7                   result          OUT NOCOPY VARCHAR2 )
 8   IS
 9
10      L_batch_id                 gl_je_batches.je_batch_id%TYPE;
11      l_boolean_validity_check   BOOLEAN;
12
13 BEGIN
14   IF ( funcmode = 'RUN'  ) THEN
15
16
17      l_batch_id := wf_engine.getItemAttrNumber ( itemtype, itemkey, 'BATCH_ID');
18
19
20      l_boolean_validity_check   := my_custom_validity_check (l_batch_id);
21
22      IF l_boolean_validity_check THEN
23       result := 'COMPLETE:Y';
24      ELSE
25       result := 'COMPLETE:N';
26      END IF;
27
28   ELSIF (funcmode = 'CANCEL') THEN
29     NULL;
30   END IF;
31 END is_je_valid;
```


© Copyright 2008 by Daniel Stober

In the listing above, my changes to the procedure begin on line 10. I've declared two variables, one to hold batch_id from the batch being validated, and one to hold the Boolean returned by my custom validation function. On line 17, the procedure calls the getItemAttrNumber to get the batch_id associated with this process instance. Then, the procedure uses that batch_id to make the call to my custom function on line 19 and assign the value returned to the variable l_boolean_validity_check. The contents of that unseen function are not important to this example. It can include any validation algorithm that can be written in PLSQL. The important aspect of this example is to understand how a customization fits into the scheme of this Workflow.

Finally, on line 22, the procedure checks the l_boolean_validity_check variable to determine whether to return "Yes" or "No." On lines 23 and 25, you'll see the possible statements that will be returned, but notice that they do not return "Yes" or "No." Instead, they return the lookup code associated with each of those two results, "Y" and "N". Voilá! I have customized the procedure.

By the way, don't forget about the other procedures in the package. If I am customizing is_je_valid, I still must place those other procedures into the package.

SET WAIT DATE

In this example, I've written a procedure that institutes a complex timeout on an approval notification. The functionality is accomplished by loading the date and time that the notification should expire into an item attribute. The procedure pictured in figure 25 illustrates what the Workflow Process might look like.

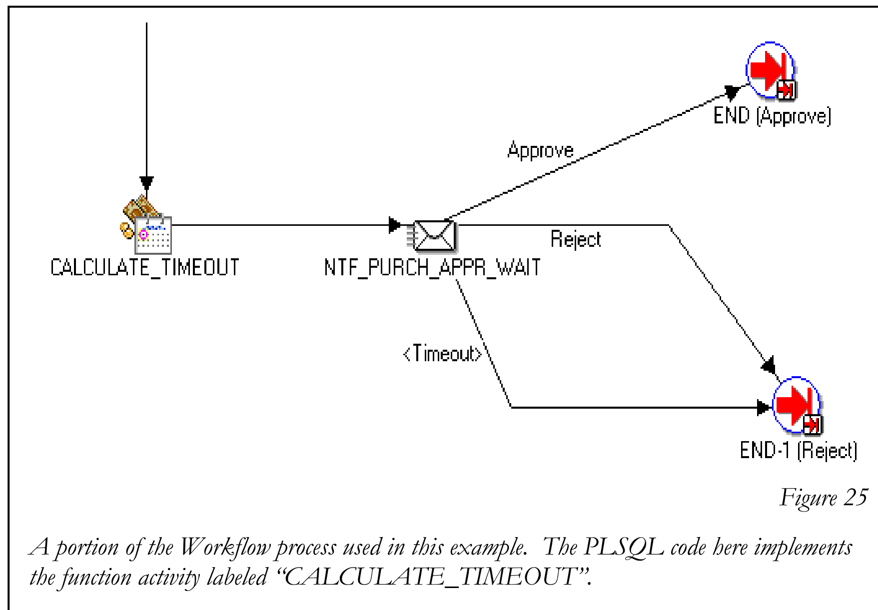

Figure 25

A portion of the Workflow process used in this example. The PLSQL code here implements the function activity labeled "CALCULATE_TIMEOUT".

The timeout date and time is calculated and loaded into item attribute in the activity node prior to the notification node.

The PLSQL procedure that follows is the code that implements the "CALCULATE_TIMEOUT" node. The most significant thing I am illustrating here is how you can use an item attribute to communicate details back to the Workflow Engine. Unlike the previous examples, this procedure does not pass anything in the ResultOut parameter beyond the usual "COMPLETE" status code.

The basic requirement here is that the notification should timeout one month after the process is launched. However, if that one month later date falls on a weekend, the timeout will be extended until the following Monday.

Here's the procedure:

```
1 PROCEDURE setWaitDate
2   (   itemtype         IN  VARCHAR2
3   ,   itemkey          IN  VARCHAR2
4   ,   actid            IN  NUMBER
5   ,   funcmode         IN  VARCHAR2
6   ,   resultOut        OUT VARCHAR2        )
7
8 IS
```

```
 9    l_ProcLaunchDate   DATE;
10    l_OneMonthLater    DATE;
11 BEGIN
12    -- Get the process launch date for the item_type and item_key that
13    -- were passed in.  Query the wf_items table directly.
14    SELECT wi.begin_date
15      INTO l_ProcLaunchDate
16           FROM wf_items   wi
17     WHERE wi.item_type = itemtype
18       AND wi.item_key  = itemkey;
19
20    -- Calculate the one month later date
21    l_OneMonthLater  := ADD_MONTHS(l_ProcLaunchDate, 1);
22
23    -- make allowance for weekends
24    IF TO_CHAR(l_OneMonthLater, 'D')    = 7 THEN          -- Saturday
25      l_OneMonthLater        := l_OneMonthLater +2;
26    ELSIF TO_CHAR(l_OneMonthLater, 'D') = 1 THEN   -- Sunday
27      l_OneMonthLater        := l_OneMonthLater +1;
28    END IF;
29
30    -- Write the calculated value to the attribute “WaitUntilDate”
31    setItemAttrDate (
32        itemtype         => itemtype
33    ,   itemkey          => itemkey
34    ,   aname            => 'WAITUNTILDATE'
35    ,   avalue           => l_OneMonthLater);
36
37   -- Mark the activity status COMPLETE.  No result code passed back.
38   resultOut := wf_engine.eng_completed;
39
40 EXCEPTION
41   WHEN OTHERS THEN
42     wf_core.context ('djs_wf_demo', 'setWaitDate'
43            , itemtype, itemkey, to_char(actid), funcmode
44             , substr(sqlerrm, 1, 100) )
45     -- Mark the activity status ERROR.
46     resultOut := wf_engine.eng_error;
47
48 END;
```

At lines 9 and 10, the two variables needed for the logic are declared. At line 14 is a SELECT statement that queries wf_items using the itemkey passed into the procedure to determine the date that the process was launched. The value found in the query gets loaded into one of the two local variables. While I prefer using APIs over querying the Workflow tables whenever possible due to their efficiency, I didn’t use an API for this because there are no public APIs to derive this datum.

At line 21, is the calculation of the one month later date. It employs a couple of Oracle’s built-in functions to derive Midnight of the date which is exactly one month later than the process launch date. The IF structure between lines 24 and 28 tests the newly derived date to see if it falls on a Saturday or a Sunday, and then adds two days or one as necessary to arrive at Monday.

Finally, at line 31, the procedure loads that value into the item attribute that will be used for the notification timeout, and then marks the process status as complete in line 38. Notice that the resultOut does not include a component for result_code is there is not a need for one in this activity. There is no transition from this activity that depends on a result code. Also, notice that, unlike the first two examples, I used the “eng_completed” constant here instead of the string “COMPLETE”. Both approaches yield the same outcome.

REPEAT_RESULT

This example demonstrates application of your knowledge of the Workflow tables to create a function.

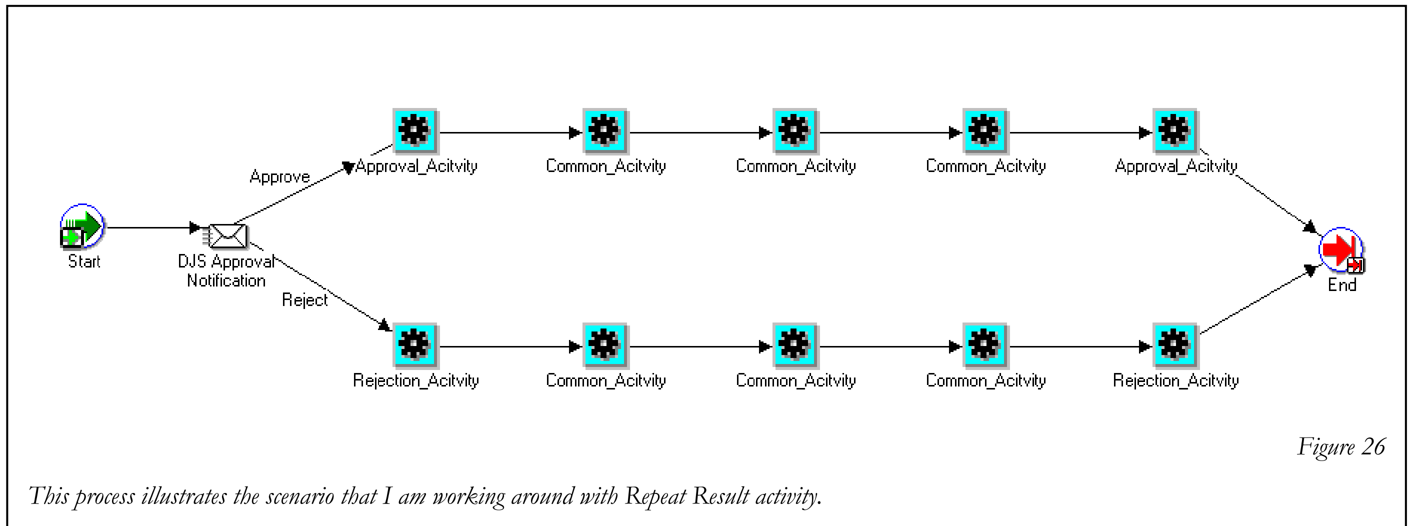

Figure 26

This process illustrates the scenario that I am working around with Repeat Result activity.

I've created the process above to help illustrate the problem I'm trying to resolve with this custom code. This is not an actual Workflow, but the scenario I'm describing is a fairly common dilemma. The process begins with a notification which can result in one of two results, "approve" or "reject". Immediately after the notification, the process branches into two different lines depending upon the result and then transitions to an activity that is specific to the result. If the notification was approved, then the Workflow Engine executes an activity that only needs to be performed in cases of approval. I've labeled it "Approval Activity" in Figure 26 above. Similarly, if the notification is rejected, then there is a "Rejection Activity" that is specific to that result.

Following those, are three activities which are common to either result; regardless of whether or not the notification was approved, these three activities need to be executed. After those three common activities, comes another pair of activities that are specific to the result of the approval notification.

In order to maintain the logic, I had to keep the two flows in the process separate until the second set of activities that are dependent upon the result, and this means that I have been forced to include each of the three common activities twice to allow them to be included in both lines of logic.

In this process, it's still pretty simple, so it's probably not too burdensome to include each those common activities twice. But what if the notification had three possible results? What if there were more common activities between the two activities that are specific to the result? We've all seen processes like this, and probably some of you have designed such a process! The process diagrams can get very complicated, very cluttered, and very difficult to follow very quickly.

To solve this, I wanted to design a process that would split for the activities that are specific to the result, but rejoin for the common activities, so that each of them appears in the diagram only once. To accomplish this, I needed to create an activity that would re-split the process flow based upon the results of an earlier activity. Figure 27 below shows how this might work.

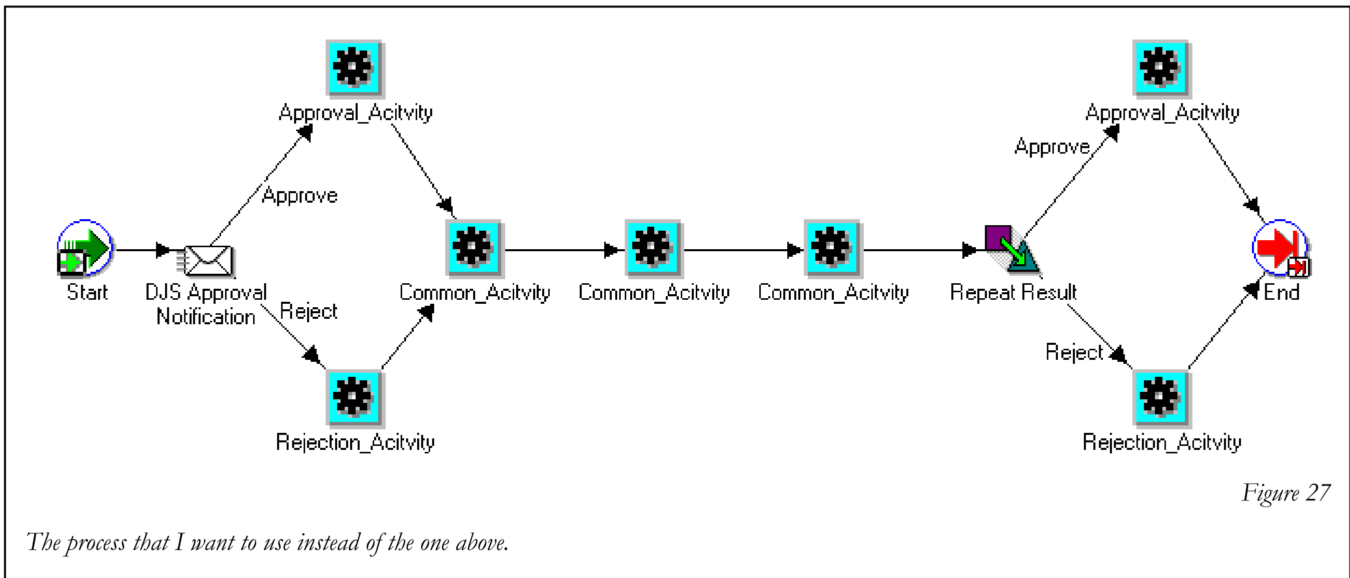

Figure 27

The process that I want to use instead of the one above.

Notice the new "Repeat Result" activity incorporated into this process. That activity looks back to an earlier activity node and returns the same result as the earlier node. In this process, the new activity is repeating the result from the "DJS Approval Notification" node. This allows me to re-split the process flow.

I designed Repeat Result to be flexible enough that I can use it in other processes, too. The activity utilizes an activity attribute to hold the instance label of the activity whose result should be repeated. Instance label uniquely identifies any node within a process, so that alone is enough to tell the Workflow Engine which activity result should be repeated.

(In order to keep the process fairly straightforward, I limited the usage of this activity to another activity within the same process. For cases, where a process instance transitions through multiple processes, "instance label" alone would not be a unique identifier among all nodes visited in a process flow. After reading the code, you'll be able to see how you to extend this function activity to permit that.)

So here is the code that implements my "repeat result" function activity.

```
 1 PROCEDURE djsResultToAttr
 2    (   itemtype  IN   VARCHAR2
 3    ,   itemkey   IN   VARCHAR2
 4    ,   actid     IN   NUMBER
 5    ,   funcmode  IN   VARCHAR2
 6    ,   resultOut OUT  VARCHAR2 )
 7 IS
 8
 9     l_instance_label_attr          VARCHAR2(30);
10     l_process_item_type            wf_process_activities.process_item_type%TYPE;
11     l_process_name                 wf_process_activities.process_name%TYPE;
12     l_process_version              wf_process_activities.process_version%TYPE;
13     l_instance_id                  wf_process_activities.instance_id%TYPE;
14     l_activity_label               VARCHAR2(30);
15     l_activity_result_code         wf_item_activity_statuses.activity_result_code%TYPE;
16
17     e_activity_not_in_process      EXCEPTION;
18     e_no_completed_activity_found  EXCEPTION;
19
20 BEGIN
21
22      IF funcmode = wf_engine.eng_run THEN
23
```

```
24        -- Read the value contained in the attribute
25        l_instance_label_attr   :=  wf_engine.getActivityAttrText
26                                    (   itemtype            =>  itemtype
27                                    ,   itemkey             =>  itemkey
28                                    ,   actid               =>  actid
29                                    ,   aname               =>  'INSTANCE_LABEL'
30                                    ,   ignore_notfound     =>  TRUE             );
31
32        -- Get information about the process that the activity calling this is a part of
33         SELECT wpa.process_item_type
34            ,   wpa.process_name
35            ,   wpa.process_version
36           INTO l_process_item_type
37            ,   l_process_name
38            ,   l_process_version
39           FROM wf_process_activities wpa
40          WHERE wpa.instance_id = actid;
41
42       BEGIN
43
44        -- Then, get the instance_id of the activity whose value should be repeated
45         SELECT wpa.instance_id
46           INTO l_instance_id
47           FROM wf_process_activities wpa
48          WHERE wpa.process_item_type   = l_process_item_type
49            AND wpa.process_name        = l_process_name
50            AND wpa.process_version     = l_process_version
51            AND wpa.instance_label      = l_instance_label_attr;
52
53      EXCEPTION
54          WHEN no_data_found THEN
55          -- Either the activity attribute attribute does not exist or it contained a string that
56          -- is not an instance label in this process
57
58             RAISE e_activity_not_in_process;
59      END;
60
61      BEGIN
62          SELECT activity_result_code
63            INTO l_activity_result_code
64            FROM wf_item_activity_statuses
65            WHERE process_activity = l_instance_id
66              AND item_type        = itemtype
67              AND item_key         = itemkey
68              AND activity_status  = wf_engine.eng_completed;
69
70      EXCEPTION
71          WHEN no_data_found THEN
72
73             RAISE e_no_completed_activity_found;
74      END;
75
76      resultOut := wf_engine.eng_completed || ':' || l_activity_result_code;
77
78   END IF; -- funcmode = wf_engine.eng_run
79
80 EXCEPTION
81    WHEN e_activity_not_in_process THEN
82
83        wf_core.context ( 'djsResultToAttr' , NULL
84            , itemtype , itemkey , TO_CHAR(actid), funcmode
85            , 'No activity with label ' || NVL(l_instance_label_attr, '<null>') || ' found in this process.');
86        RAISE;
87
88    WHEN e_no_completed_activity_found THEN
89
90        wf_core.context ( 'djsResultToAttr' , NULL
91             , itemtype , itemkey , TO_CHAR(actid), funcmode
92             , 'No completed activity named ' || l_activity_label);
93        RAISE;
94
95    WHEN OTHERS THEN
96
97        wf_core.context ( 'djsResultToAttr' , NULL
98                 , itemtype , itemkey , TO_CHAR(actid), funcmode
99                 , SUBSTR(SQLERRM, 1, 100));
00
01        RAISE;
```


© Copyright 2008 by Daniel Stober

```
02
03 END djsResultToAttr;
```

The procedure accomplishes its goal in five steps:

1. Read activity attribute to get the instance label of the function activity to be repeated. This procedure does this via an API call at line 25 I used the getActivityAttr… API to read the value of the activity attribute named "INSTANCE_LABEL".

   In this case, I included the optional parameter to "ignore_notfound". This means that the API will not raise an error if the activity does not include that attribute, it will just return null. Eventually, if this condition exists, it will result in an error in Step 3, but approach allows me to combine error handling.

2. Derive the name of the current process and version. At line 33, I query wf_process_activities to gather the information about the current process. The program needs to know the name and item type of the process that contains the activity that is calling this code because it is going to have to find the function activity that needs to be repeated within the same process. The program also needs to know the version number of the process because it will have to derive the instance id of the target process for the next two steps.

   Fortunately, the Workflow Engine has passed the instance_id of the function activity that called this code into this procedure, so querying this information involves only a single table.

3. Get the instance id of the target function activity from the version of the process. As I showed during my review of the tables, instance_id is the unique identifier within wf_process_activities, and all records within wf_item_activity_statuses use it in a foreign key relationship. Within wf_process_activities, a unique combination of instance_label and version within any process_name/process_item_type combination constitutes a unique record, so the program has all of the information necessary to complete this step.

   I've included error handling around this step in case the developer placed a value into the attribute that tests to ensure that an activity with the label specified by the label exists in the process. Since the activity attribute is only a text attribute, virtually any string could be entered, so the error handler here makes sure that attribute contained the instance label a valid activity. If the attribute value is null or does not exist, those will be handled here, too.

4. Get the result. Finally, at line 62, the process can query wf_item_activity_statuses to see what the activity result was in the target activity. Notice that the WHERE clause includes a clause to select the result only if the activity has been completed. The "no_data_found" handler in this case is here in case the target activity has not been completed yet. In the example process shown in Figure 27, it would not be possible for the process flow to transition to "RepeatResult" unless the target activity had been completed already. However, I wrote this procedure to be robust enough that it could be used in other processes, so it is necessary that I test to see if the activity is actually completed. (Actually, even this is not enough – I should set this activity's status to "DEFERRED" and wait for the other activity to be completed. But, since the procedure is already 103 lines long, and I wanted to keep the example manageable, I left it as is.)

5. Repeat the result. Finally, the procedure returns the same result as the target activity.

UNIT IV: OTHER CODE TO WRITE FOR WORKFLOW

Chapter 8: A CONCURRENT PROGRAM

The following program is different from the others in this paper. While the others are called by the Workflow Engine, this one is written to be called by the Concurrent Manager. The purpose of this procedure is to remove any individual notification preference for all roles so that they will receive notifications in the style of the company default.

Earlier on in the paper, I mentioned that roles would not be a part of the discourse here because the users and roles have been integrated into the applications foundation (“FND”) tables. However, I want to include this program listing because it’s a solution I wrote to resolve a serious error that pops up with the notification mailer.

Under version 11.5.10.2 (and I think it’s been this way since 11.5.9), the notification mailer tries to help you out whenever it attempts to email a notification and the notification fails. The help that it provides is that it sets the notification preference of the role to “DISABLED”.

The assumption is that, if the email failed, then there must be something wrong with the email address, so it’s better to disable the role rather than continue to try to send other messages. Once the user corrects the email address, they also can reset the preference and resume receiving emails.

The problem with that assumption is that bad email addresses is not the only reason that notifications fail. Anyone who has experienced notification mailer issues from time to time knows that this featurecan result in a lot of email preferences changed. Once the mailer is restarted, contacting each of the users and having them reset their own preference is time consuming and inefficient. That’s where this procedure comes in handy.

The algorithm for the notification preference is to check for an individual preference. If it exists, use it, otherwise use the company default preference. When the notification mailer disables a user preference, it inserts an individual preference for the user that it is disabling. My program identifies all users who have an individual preference record and removes it.

It’s a simple program that can be run from the Concurrent Manager and it completes very quickly. I caution you about one thing: the two APIs used here are not public APIs. I had to get permission from Oracle support to use them in our production environment. But , if this is a problem that has bedeviled you, you might consider getting permission, too.

```
 1 PROCEDURE reset_notification_prefs
 2     (   p_errbuff        OUT VARCHAR2
 3     ,   p_retcode        OUT VARCHAR2)
 4
 5 AS
 6
 7    l_rec_count         NUMBER    :=0;
 8    l_err_count         NUMBER    :=0;
 9
10    l_savepoint_name    VARCHAR2(25);
11    l_err_loc           VARCHAR2(100);
12
13 BEGIN
14
15    FOR prefs_rec IN (SELECT *
16                        FROM applsys.fnd_user_preferences
17                       WHERE preference_name   = 'MAILTYPE'
18                         AND module_name       = 'WF'
19                         AND user_name         != '-WF_DEFAULT-'
20    ) LOOP
21
22        BEGIN
23
24            v_savepoint_name := prefs_rec.user_name;
25
26            SAVEPOINT l_savepoint_name;
27
28            l_err_loc       := 'API calls on user ' || prefs_rec.user_name;
29
```

```
30            fnd_preference.remove ( p_user_name     =>  prefs_rec.user_name
31                                  ,   p_module_name   =>  prefs_rec.module_name
32                                  ,   p_pref_name     =>  prefs_rec.preference_name );
33
34            fnd_user_pkg.user_synch(prefs_rec.user_name);
35
36            l_rec_count     :=     l_rec_count     + 1;
37
38        EXCEPTION
39            WHEN OTHERS THEN
40
41                ROLLBACK TO l_savepoint_name;
42
43                l_err_count     :=     l_err_count     + 1;
44                fnd_file.put_line(fnd_file.log, 'Error at ' || l_err_loc || ' > ' || SUBSTR(SQLERRM, 1, 500));
45
46        END;
47    END LOOP;
48
49    COMMIT;
50
51    fnd_file.put_line(fnd_file.log, l_rec_count || ' user preference records removed.');
52
53
54    IF l_err_count   = 0 THEN
55
56        p_errbuff   := 'SUCCESSFUL COMPLETION. ' || l_rec_count || ' user preference records removed.';
57        p_retcode   := 0;
58
59    ELSE
60
61        p_errbuff   := 'COMPLETION WITH ERRORS. '
62                    || l_rec_count || ' user pref records removed '
63                    || l_err_count || ' errors.';
64        p_retcode   := 1;
65
66    END IF;
67
68 EXCEPTION
69    WHEN OTHERS THEN
70
71        ROLLBACK;
72
73        p_errbuff   := 'ERROR OCCURRED.  See log for details';
74        p_retcode   := 2;
75
76 END reset_notification_prefs;
```

The procedure begins with the two outbound parameters required for a concurrent program, and then four local variables. At line 15, the cursor that finds all of the preference records that need to be deleted begins. Notice that the notification preference is stored in fnd_user_preferences and not a workflow table.

The first of the two API calls is at line 30, and this one actually removes the preference record. The second API call, at line 34, performs the synchronization which propagates this change to all of the other older tables

That's it. Very simple. The rest of the program is just error handling and return values.

Chapter 9: SELECTOR FUNCTIONS AND CALLBACK FUNCTIONS

A selector function is a PLSQL procedure that is used by the Workflow Engine to determine what process should be run when none was specified. A selector function is specific to the entire item type. Earlier on in the paper, back on page 24, we saw that the APIs createProcess and launchProcess allow you to launch a process simply by specifying an item_type and providing an item_key – just

two parameters.. The Workflow Engine determines which of the processes within that item_type should be run based upon the value returned by the item_type's selector function[6].

A callback function is code that executes before a notification is delivered and again after it is responded to. The function allows you, as the developer, to write code that tests the state of variables in session before delivering the notification to ensure consistency (in case of a loss of state) and update them if necessary. If you create your own notifications, you also can use the callback function to perform updates to item attributes based upon values the user entered while responding to the notification.

The tricky thing about these functions is that the *same* PLSQL procedure is used for the selector and callback functionality. When it calls the procedure, the Workflow Engine uses one of the parameters to tell the code about the purpose of the call. This means that when you develop a procedure to take advantage of this functionality, your code must test the value passed within that parameter to determine the reason that the call is being made. We'll explore that momentarily.

When you create a new item_type, the selector function is not a required element, but a selector function must be included if a call to one of the APIs listed above does not supply a value for the "process" parameter. This is true even in the case of items which have only one process, although in this section I'll show you how to write a generic function that will work whenever there is only one process in an item_type.

Also, be aware that although "Selector Function" is an optional property of an item type, if one has been defined. Then you must ensure that the procedure exists, even if you are specify a process name while launching the Workflow. The procedure is still called before and after every notification node.

First, let's take a look the problem that we are trying to avoid:

```
ORA-20002: 3108: Root process could not be determined for item 'DJS_DEMO/123'.  There may be a problem with
the item type selector function, or you may be required to supply a starting process name.
```

The error message above is what you'll receive when you call the wf_engine.createProcess API for a item_type that does not have a selector function defined and you don't specify the process name. You'll also see that error message if there is a selector function, but it does not return the name of a valid, runnable process. The solution is to create a selector function, so the Workflow Engine will know which process to run.

The call signature for a selector function looks like the one shown below:

```
PROCEDURE my_selector
    (   item_type     IN     VARCHAR2
    ,   item_key      IN     VARCHAR2
    ,   activity_id   IN     NUMBER
    ,   command       IN     VARCHAR2
    ,   resultout     IN OUT VARCHAR2 );
```

Once again, even though the implementation is called a selector function, the PLSQL structure that you will use is a procedure. Of course, the discussion about the spelling of the names of parameters from page 36 is still relevant here. In the example above, I've spelled the parameters the way that the documentation specifies, and I'll use those names in my discussion to distinguish between them. As I've already mentioned, whether you spell them that way in your coding is up to you.

---

[6] Oracle Workflow Developers Guide, version 2.5, P4-4

If the parameter list for a selector function looks remarkably similar to the requirements for a function activity, that's because internally, the Workflow Engine uses the same dynamic PLSQL call to execute both of them. The procedure still requires four inbound parameters and one outbound. For a selector function, "activity_id" is not used, so the value of that parameter will be null on every call, but it still must be included in the procedure.

Of the other parameters, "command" is the most noteworthy in this discussion. Since the same procedure serves multiple purposes, selector function and callback function, the value passed in that parameter can be used by the program to distinguish for which objective the call to the procedure was made. In the case of a selector function, the Workflow Engine will pass the constant value wf_engine.eng_run ("RUN") into the procedure. This means that when you write your procedure, the portion of the procedure which is devoted to selector logic should be enclosed within an IF statement that tests for the eng_run value in the "command" parameter:

```
IF command = wf_engine.eng_run THEN
        < selector logic goes here >
END IF;
```

The other two inbound parameters, "item_type" and "item_key", can be referenced by the selector function as necessary, depending upon the circumstances. For example, if you create a selector function procedure whose use will be dedicated to one particular item_type, it is not necessary to reference the "item_type" parameter in the procedure. You just have to make sure that only the intended item_type is set up to use the procedure. On the other hand, if you designed your Workflow item_types so that several of them utilize the same procedure as a selector function, then you will definitely want to reference the "item_type" parameter in the procedure so that you can return the correct procedure for the desired item_type.

Finally, we come to the sole outbound parameter. Use the "resultOut" parameter to assign a string that contains the name of the process that should be run when an item in the item_type is created. When you write the procedure, the procedure should finish by assigning the internal name of the procedure that should be started to that parameter.

(You may have noticed that this parameter should be of an IN OUT subtype. When the procedure is called in a role of a selector, no data is passed inward to the procedure within this parameter, so this parameter operates as outbound only. However, since the same procedure serves other functionality, too, you should follow the documentation, and create this parameter as IN OUT.)

To demonstrate how this can work, I've written a procedure that works as a generic selector function for any item type that contains only one process:

```
 1 PROCEDURE generic_selector
 2   (   item_type     IN     VARCHAR2
 3   ,   item_key      IN     VARCHAR2
 4   ,   activity_id   IN     NUMBER
 5   ,   command       IN     VARCHAR2
 6   ,   resultout     IN OUT VARCHAR2 )
 7 IS
 8
 9   l_process_name    wf_activities.name%TYPE;
10   l_process_count   PLS_INTEGER     :=0;
11
12 BEGIN
13    IF command = wf_engine.eng_run THEN
14
15       FOR proc_rec IN (SELECT name
16                          FROM wf_activities    wa
17                         WHERE wa.item_type   = item_type
18                           AND wa.type        = 'PROCESS'
19                           AND runnable_flag  = 'Y'
20                           AND SYSDATE BETWEEN wa.begin_date
21                                        AND NVL(end_date, SYSDATE)
22        ) LOOP
```

```
23
24          l_process_name  := proc_rec.name;
25          l_process_count := l_process_count +1;
26
27       END LOOP;
28
29       IF l_process_count = 1 THEN
30        -- Only one process in this item_type, so return its name
31
32          resultout := l_process_name;
33
34       ELSE
35
36          resultout := NULL;
37
38       END IF;
39
40    END IF;
41
42 END generic_selector;
```

This process checks within the item type for which the code is called to look for process activities. If it finds exactly one process in the item type, then it returns the name of that process; otherwise, it returns null. The procedure returns the data by means of its single outbound parameter, “resultOut”.

Looking at the code, starting at line 13, you’ll see an “IF” statement which restricts execution of all of the following statements through to line 40, to cases where the Workflow Engine has passed in a value of “RUN”. Remember, since the same procedure is used both as a selector function and as a callback function, the Engine will pass in “RUN” when calling the code in the capacity of a selector function. In the case of this procedure, I am using the constant “eng_run” from the wf_engine package instead of hard-coding the value.

Beginning at line 15, the procedure employs a cursor loop driven by a query of all process activities in the item type. Within the loop body, the procedure assigns the internal name of the process from the cursor’s current record to a variable and increments a counting variable. After the loop, at line 29, the logic tests to see if exactly one record was found. If so, then the l_process_name variable will contain the internal name of that process. At line 32, that internal name is assigned to the ResultOut parameter, so that it can be returned to the calling program as the result of the selector function.

On the other hand, if the loop found more than one process activities or no process activities at all in the item type, then the procedure will set the resultOut to null. The calling program would then raise an error because it would not be able to launch a process.

Chapter 10: CODING FOR A DOCUMENT-TYPE ATTRIBUTE

WHY USE A DOCUMENT-TYPE ATTRIBUTE?

Item attributes generally hold data of one of the following types: number, date, or character. In most cases these three choices are more than sufficient for a Workflow process, but they do have limitations.

One of those limitations is that attributes of character types cannot exceed 4,000 characters in length. The reason for this limitation is that these values are stored in a VARCHAR2 field in wf_item_attribute_values, and VARCHAR2 fields in database tables are limited to 4,000 characters.

Two re

1) The

(ie: it e

2) The

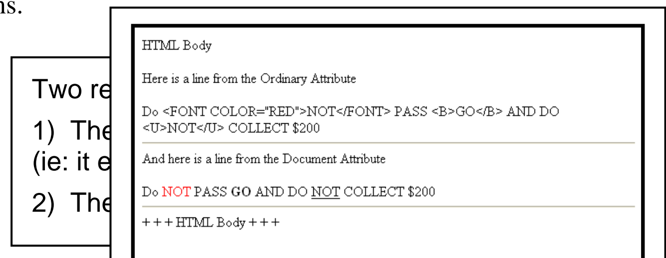

Figure 28

An example message containing two attributes: One table-based and one document-type. The two attributes contain the same text, but the first one displays the HTML tags while the document attribute resolves them as I had hoped.

© Copyright 2008 by Daniel Sto

The other limitation on regular, table-based item attributes is that regular text attributes cannot contain HTML markup. Even if the attribute is being used to feed a message attribute that is on the HTML message tab, the value of the attribute itself is limited to text only.

The solution to overcome both of these limitations is to use a Document type attribute. As you may have inferred, the value of a document attribute is not stored in a table at all – it is derived at runtime as the return parameter of a PLSQL procedure. This means that you can include text strings that far exceed the limitation imposed upon fields stored in tables. With a VARCHAR return type, your attribute can stretch up to 32767 characters, and if you use a CLOB return type, it is virtually unlimited. And, of course, the only way that you can include HTML tags into an attribute is to use a document type attribute.

SETTING UP A DOCUMENT-TYPE ATTRIBUTE

Once you've decided that using a document-type attribute is appropriate, it's time to set it up. The steps that you'll need to perform are:

Steps performed in the Workflow Builder…

1. Create a Document-type attribute for a message
2. Define a default value for the message which points to the code you will write in step 4
3. Include the attribute in the new message

Step performed in a text editor and compiled n the database…

4. Write a procedure to be called by your attribute and compile

After this, this message can be inserted into a notification, just like any other message. Of course, I you choose you may perform the last step before the first three because it really doesn't matter. In an actual development cycle, I generally do the Workflow Builder steps first and then write the code, but I chose the order above for this paper because I want to devote most of the text in this section to writing the PLSQL for a document attribute. The order above allows me to dispense with the formalities and then stay focused on the heart of this topic.

STEP ONE

Figure 29

Creating a document type attribute

In the Workflow Builder, create an attribute for the message.

In the properties of the newly created attribute, locate the drop down box labeled "Type", and choose "Document." In the example pictured in Fig 29, the attribute with the display name "My Document Type Attribute" has an internal name of "DOCUMENT_ATTRIBUTE".

STEP TWO

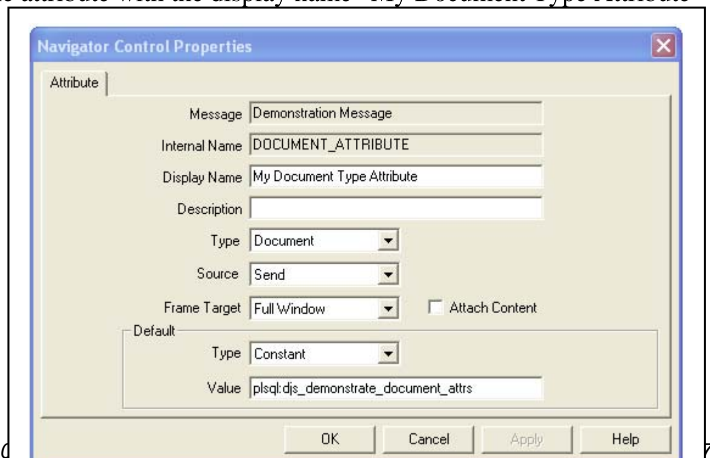

Figure 30

Steps one and two: Define the attribute with type "Document" and assign a default value

Still in the properties window, take the name of the procedure you will create in Step Four, and place it in the field labeled, “Default”. The name of the code should be preceded by “plsql:” (include the colon.). In general, the default will look something like this

```
plsql:my_package_name.my_proc_name
```

In the examples pictured, I am creating a procedure called “djs_demonstrate_document_attrs” to populate the value of this document attribute. I place the name of that procedure in the field for the default value of the attribute, using the format shown above, with a leading “plsql:” (Don’t forget the colon!).

STEP THREE

Within the text of the message you created, include the new attribute value. Remember, use the internal name of the attribute.

In this example, I've included text above and below the attribute to make it more obvious when you see the notification.

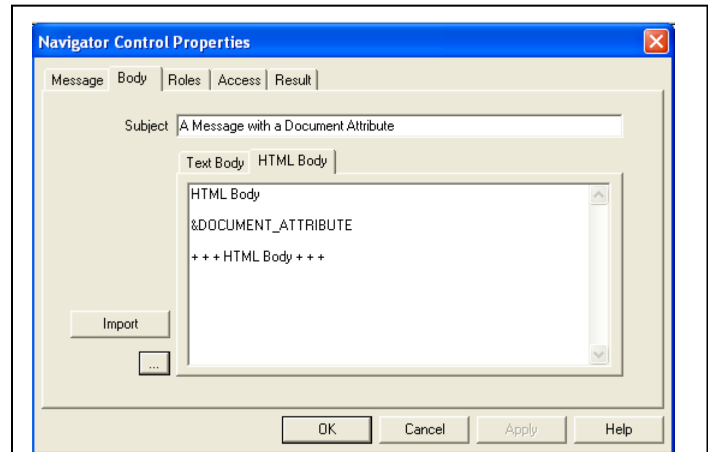

Figure 31

Placing the document attribute into a message

STEP FOUR

Write the PLSQL procedure that you will use to populate the attribute. This will be the most difficult step. The call signature that you should use for your code should look like this:

```
PROCEDURE your_custom_procedure
    (   document_id       IN      VARCHAR2
    ,   display_type      IN      VARCHAR2
    ,   document          IN OUT NOCOPY VARCHAR2
    ,   document_type     IN OUT NOCOPY VARCHAR2);
```

(or you could use CLOB instead of VARCHAR2)

The entire stream of text that will be written to the attribute should be assigned to the "document" IN OUT parameter.

THE PLSQL PROCEDURE

Writing the PLSQL code to be referenced by a document-type attribute is similar to the process outlined above for PLSQL for the other Workflow activities. Similar, but not identical.

The first difference you'll note is that the call signature is different. The call specification for normal Workflow function activity calls and selector functions provides four IN parameters that supply the procedure with lots of information about the calling activity node. The document-type attribute call signature, in contrast, has no parameter to tell you what item_type and item_key are associated with the process that is calling the code. The document-type attribute procedure makes the programmer work a little harder to get the same information!

But before I get into techniques for that, let's just set up a simple one to see how it works.

```
1 PROCEDURE djs_demonstrate_document_attrs
2     (   document_id                       IN      VARCHAR2
3     ,   display_type                      IN      VARCHAR2
```

 © Copyright 2008 by Daniel Stober

```
 4     ,   document                        IN OUT NOCOPY VARCHAR2
 5     ,   document_type                   IN OUT NOCOPY VARCHAR2)
 6 IS
 7
 8    l_document_html VARCHAR2(32767);
 9
10 BEGIN
11
12     l_document_html :=
13            '<H1>These are the values in the<BR>WF Standard
14         || '<BR>Comparison Lookup Type:</H1>'
15         || '<TABLE BORDER="1">';
16
17     FOR lookup_rec IN ( SELECT lookup_code, meaning
18                        FROM wf_lookups
19                        WHERE lookup_type = 'WFSTD_COMPARISON' ) LOOP
20
21         l_document_html := l_document_html
22             || '<TR><TD>'  || lookup_rec.lookup_code
23             || '</TD><TD>' || lookup_rec.meaning
24             || '</TD></TR>';
25
26     END LOOP;
27
28     l_document_html := l_document_html
29           || '</TABLE>';
30
31     document := document
32            || l_document_html;
33
34 END djs_demonstrate_document_attrs;
```

In this example, I am building an HTML table that contains all of the lookup codes that make up the Standard Comparison lookup type and their meanings. Because the procedure's name is "djs_demonstrate_document_attrs", it will implement the document attribute that I set up in Figure 30.

From lines 2 through 5, you see the four parameters that are used for a document-type attribute, and then the local variable that the procedure will use to build the value of the attribute is declared at line 8. Ultimately the value of the attribute will be assigned to the parameter "document". I could have built the attribute value directly in the parameter, but I chose to use a separate variable in the interim and then assign the final value back to the parameter at line 31.

At line 12, the procedure creates an <h1> label header for the table, and then opens the table. Following that is a cursor of the relevant records, and PLSQL code that builds the HTML markup, row by row. Afterward, the closing table tag is added, and the local variable then contains all of the markup that will populate the attribute.

Finally, at line 31, the procedure assigns the local variable to the parameter. You may notice that the assignment statement re-assigns the existing value of the document parameter to itself before appending the markup contained in the local variable. That is because that parameter is defined as "IN OUT". Be sure to do it that way or any information that is situated in the body of the message prior to the location of the document attribute will be lost from the message.

Also, I did not include an error handler in this procedure. That decision was only for the purposes of this example. We have not yet covered how to derive the item_type and item_key, so I omitted it from this example.

So, with all of these pieces in place, Figure 32 shows how the notification looked when it was emailed to me.

HTML Body

These are the values in the WF Standard Comparison Lookup Type:

EQ	Equal
GT	Greater Than
LT	Less Than
NULL	Null

+ + + HTML Body + + +

Figure 32

The notification that was sent using the attribute setup and PLSQL code featured above.

THE DISPLAY_TYPE PARAMETER

With that first example out of the way, let’s examine how a couple of the other parameters can impact the PLSQL procedure. Following this section, I’ll show you how to use the “document_id” parameter to derive the item_type and item_key. But first, in this section, I’ll show you effect of the “display_type” parameter.

In the first example, when creating the message, I stuck to the “HTML Body” tab in the Workflow Builder. I also made sure that my notification preference was set up to receive HTML emails. This was intentional – if I had received a text based notification, then my example would not have worked. To see what I mean, take a look at Fig 33 at right.

```
Text Body

<H1>These are the values in the<BR>WF Standard<BR>Comparison Lookup
Type:</H1><TABLE BORDER="1"><TR><TD>EQ</TD><TD>Equal</TD></TR><TR><TD>GT</TD><TD>
Greater
Than</TD></TR><TR><TD>LT</TD><TD>Less
Than</TD></TR><TR><TD>NULL</TD><TD>Null</TD></TR></TABLE>

+ + + Text Body + + +
```

Figure 33

Including HTML markup in a Document type attribute does not have the desired impact when the user receives text notifications.

```
 1 PROCEDURE djs_demonstrate_document_attrs
 2    (   document_id                    IN     VARCHAR2
 3    ,   display_type                   IN     VARCHAR2
 4    ,   document                       IN OUT NOCOPY VARCHAR2
 5    ,   document_type                  IN OUT NOCOPY VARCHAR2)
 6 IS
 7
 8     l_document_html VARCHAR2(32767);
 9     l_document_text VARCHAR2(32767);
10
11 BEGIN
12
13     l_document_html :=
14            '<H1>These are the values in the<BR>WF Standard'
15            || '<BR>Comparison Lookup Type:</H1>'
16            || '<TABLE BORDER="1">';
17
18     l_document_text :=
19          'These are the values in the WF Standard '
20       || 'Comparison Lookup Type: || CHR(13) || CHR(10);
21
22     FOR lookup_rec IN ( SELECT lookup_code, meaning
```

```
23                          FROM wf_lookups
24                         WHERE lookup_type = 'WFSTD_COMPARISON' ) LOOP
25
26         l_document_html := l_document_html
27            || '<TR><TD>'  || lookup_rec.lookup_code
28            || '</TD><TD>' || lookup_rec.meaning
29            || '</TD></TR>';
30
31         l_document_text := l_document_text || CHR(13) || CHR(10)
32            || RPAD(lookup_rec.lookup_code, 6)
33            || lookup_rec.meaning;
34
35    END LOOP;
36
37    l_document_html := l_document_html
38           || '</TABLE>';
39
40    IF display_type = wf_notification.doc_html THEN
41
42        document := document
43               || l_document_html;
44
45    ELSIF display_type = wf_notification.doc_text THEN
46
47        document := document
48               || l_document_text;
49
50    END IF;
51
52 END djs_demonstrate_document_attrs;
```

The changes I made here involve a new variable to hold an alternate value for the attribute to be used in cases where the user is not seeing HTML messages. The new variable is declared on line 9, and the procedure begins to fill it in at line 18. Within the cursor loop, the procedure builds a pseudo-table with the information by padding one column to align the data. By the time the cursor is complete, there are two different variables that contain essentially the same information – just in two different formats.

At line 40, the procedure finally references the display_type parameter to see which version needs to be served up. The display_type value will always be one of those two constants referenced at lines 40 and 45. Depending upon which value it is, the data contained in one of those two variables will be passed back. Simple.

THE DISPLAY_TYPE PARAMETER

Thus far, all of the examples have focused on document attributes that would send the same information to every call. However, in order to exploit the document attribute to its fullest potential, you have to be to populate it with different information for different situations.

Unfortunately, the document attribute does not permit the Workflow Engine to provide straightforward information like in the other types of code that we have examined. For this information, the developer has to work a little bit harder!

Earlier, on page 58, I showed you how you place the name of the procedure into the default value of the attribute when you set it up. In that section, however, I did not mention that you can pass additional information into the procedure by following the procedure name with a forward slash. Any data that follows that forward slash will get passed into the procedure in the document_id parameter.

For instance, this call:

```
plsql:my_package_name.my_proc_name/stuff_to_pass_in
```

Would pass the string “stuff_to_pass_in” inot the “my_proc_name procedure in the document_id parameter. You do not need to place any quotation marks around it. I know that it’s not very straightforward, but it does work!

Knowing about this capability means that you can leverage it to pass information about the Workflow process instance into the procedure. Within the Workflow Builder, you could create attributes for the item_type and item_key, and then append those attributes to the end of the call. To concatenate an attribute name, use the ampersand notation, just like you do in the message body.

There is an easier way that than, though. Oracle Workflow provides a special internal activity attribute that contains the value of the notification_id, #NID. You can design your message to pass that value into the document_id parameter. Using the notification_id alone, any other necessary information can be derived from the Workflow tables.   An example of this method using the template we showed you would look like this:

```
plsql:my_package_name.my_proc_name/&#NID
```

Notice that the procedure name is followed by, in order, a forward slash and an ampersand, and then a hash mark with the letters “NID”. The ampersand is the usual character used to designate attribute placeholders in the Workflow Builder, so, in plain English, this call says “supply the value of the internal attribute which contains the notification_id.”

The outcome of this call would be that the notification_id would be passed into the parameter document_id of the procedure called. Now that the this information is into the procedure, the program can execute this simple query and, voila, the program has the item_type and item_key information, and can begin querying the other information needed to build the notification.

```
SELECT wias.item_type, wias.item_key
  INTO v_item_type, v_itemkey
  FROM wf_item_activity_statuses wias
 WHERE wias.notification_id = document_id;
```

At any rate, that roundabout example should make the case that using #NID is, by far, the easiest, most straightforward way to achieve the desired outcome.

In closing this discussion, even though we’ve focused on a single call signature, I must alert you to the existence of a second one. The document parameter can be one of two datatypes. The majority of the discussions and examples have emphasized the VARCHAR2 implementation, but the document-type attribute also can be used to contain a CLOB if the 32,767 characters allowed for a varchar are not sufficient.

GOTCHA: Don’t forget that the “document” parameter is IN/OUT.

In the procedure that you write to implement a document type attribute, you must remember to assign the value of the parameter to itself. Similar to the plus-equals operator (“+=”) used by java programmers, your PLSQL procedure should include a statement similar to this one:

```
document := document || text_added_by_proc;
```

…or any text that appears in the message before the attribute will be dropped.

SOME LESSONS I'VE LEARNED

In my experiences with Workflow, I've found the document-type attributes very useful in trying to resolve a couple of challenges that previously had vexed me. However, the success I've achieved with these structures has not come without some accompanying headaches. I call this final sub-section of this paper "Some Lessons I've Learned", but it would not have been far-fetched to add an aside, " … The Hard Way!"

By noting these issues here, I hope that you can learn from some of my mistakes, and avoid the headaches yourself!

VALUES ARE DERIVED AT RUNTIME

After you make the Document-type attribute work successfully, the most important consideration to take into account is this: The code that populates these attributes is executed at the time the notification is opened. In a typical Workflow process, attribute values are set prior to the time the notification is sent. If there are changes to the underlying data between time that the notification is sent and the time it is read, the value contained in the attribute does not change. This would be the expected, or "normal" behavior.

But when you use an attribute of type "Document" the value of that attribute is not populated before the notification is sent, it happens at the time that the notification is opened and read. The values of document attributes are not stored in a Workflow table. They are set by a PLSQL procedure, and the values derived by that PLSQL procedure may very well be different every time it runs if the underlying data change.

NOT VERSION CONTROLLED

Oracle advertises that Workflow is version controlled: Changes made to a process definition will not impact instances of that item_type that are already running, only those that are subsequently launched. But this is *not* true when it comes to any stored procedures called by the Workflow process.

PLSQL is not version controlled, so any changes you make to the code which is called to fill a document attribute in notification will be reflected in all currently open and future notifications. For example, if the code references some new item attribute that is not present in earlier versions that may still be used in currently running processes with open notifications, you must include coding or error handling to ensure that the Workflows processes which were already running when the code was changed will continue and complete successfully.

RESTART MAILER REQUIRED

Here's one final lesson that I've learned the hard way. If you use a document-type attribute in a notification that is being sent out by the Notification Mailer, you must restart the Mailer every time you make any changes to the underlying PLSQL procedure that is called. If you don't, you'll find that your notifications are not being sent into the email system, and all of the records in wf_notifications will end up with an "ERROR" value in the mail_status field. When you check into the error, you find something this:

```
ORA-04068: existing state of packages has been discarded
ORA-04061: existing state of package "<package_name>.<proc>" has been invalidated
```

Trying to put the technical details into plain English words as much as I can, this error occurs because the session in which the Mailer runs remains connected to the database all of the time, and in the eyes of the that session, the package containing the PLSQL referenced by the document-type has become invalid.

Ordinarily, each time a user logs into the notification system, a new session is initiated. When the session begins, the newly-changed package is seen as compiled properly, and everything is fine. But since the session in which the mailer runs is connected to the database all of the time, when the CREATE OR REPLACE statement is issued to modify the procedure, that procedure gets marked as invalid in the shared pool. The only to way to rectify this is to initiate a new session for the Mailer, which is accomplished by bouncing the Mailer.

This behavior is documented in Metalink Note 303260.1.

CONCLUSION

I'd like to close this paper on a personal note. In writing this white paper, I learned one lesson that I had not anticipated: Sometimes, it's more difficult to decide what *not* to include in a paper than it is to add increasingly more subject matter into the mix.

Oracle Workflow is a huge topic with myriad possibilities for exploration. There were so many areas on which I wanted to focus, and so many issues that I wanted to incorporate, but, ultimately, I had to pare back some of my ambitions. Nothing is more wrenching to a writer than making a wholesale deletion of several paragraphs from a manuscript because they just didn't mesh with the rest of the text – even after substantial wordsmithing.

I hope that you've found the topics and issues that I chose to include here to be worthwhile, and that my overall presentation was coherent. My specific purpose when I set out was to attempt to gather Workflow–related information from several varied resources and to combine that information with my own experiences. Some of the information I've presented here is lightly documented, or the existing documentation is difficult to follow because it is mired in technicalese.

I've tried my best to be accurate throughout the paper, and to the best of my knowledge, all of the information presented is correct. However, sometimes applications and programs change. And, of course, there is always the possibility that I've made an error somewhere along the way. Please don't take anything that you read in this paper for granted until you've verified it for yourself. Always observing this practice will serve you well.

GLOSSARY

Activity – A term used to describe a broad group of objects within Workflow that can be inserted into a process as a node. It includes processes, functions, events, and notifications.

Call Signature – The list of parameters that a particular procedure or function accepts. The call signature is defined by the number, datatypes, and direction (IN, OUT, or IN OUT) of the procedure.

Callback Function – A PLSQL procedure executed before and after a notification is delivered, and again after the performer responds. The callback function function allows the developer to implement conditional logic.

Error Process – In Workflow, a process that will be launched in response to an error raised in any activity in a running process

Exception Handler – In PLSQL, a statement or group of statements the will be carried out if any error occurs during execution of the program.

Function Activity – A workflow node that represents a single step in a process and which is implemented by a PLSQL procedure (or java method)

PLSQL Procedure – A Structure in the PLSQL language characterized by a series of executable statements to accomplish a specific purpose. PLSQL procedures often are stored in the database, and can take any number of parameters, either inbound or outbound. While a procedure can have one or more outbound parameters, it is distinguished from a function by its lack of a return parameter

Process – As a noun, it can refer to two different things within Workflow. First it can represent a series of activities linked together by transitions to accomplish cohesive business purpose. Second, it can refer to an instance of an item, either running or completed, identified by a unique itemkey. When necessary to distinguish the usage, this paper will refer to the second case with the term "process instance."

Process Activity – A workflow object used to encapsulate a process

Root Process – The first process started in a single Workflow. Since one process can call another, and call another, and so on, the root process represents the process that process that got the ball rolling. The item is not considered completed until the root process reaches an END node. The name of the root process is stored in wf_items.root_activity

Runnable –An attribute of a Workflow process indicating that it can be used as the top level or root process in an item instance. The value of the attribute is stored in the "runnable_flag" field in wf_activities. Only processes designated as "runnable" are valid to be referenced in the createProcess or launchProcess API or to be returned by a selector function.

Selector Function – A PLSQL procedure which defines a default process for an item_item. If an attempt is made to launch an item, and the name of the process to be run is not specified, then the Workflow engine calls the selector function procedure to make that determination. Selector functions are optional. Although the word "function" is a part of the name, it is implement by means of a PLSQL procedure, and not a function.

 © Copyright 2008 by Daniel Stober

REFERENCES

Oracle Workflow Product Documentation
Metalink Document 67183.1

Workflow FAQ General
Metalink Document 187735.1

How to Create a New Workflow
Metalink Document 47711.1

Frequently Asked Questions on Purging of Oracle Workflow Data
Metalink Document 277124.1

White Paper :Getting Started with Workflow - Standalone 2.6.3
Metalink Document 266612.1

Workflow FAQs and Patch Information
Metalink Document 225453.1

How to Create Dynamic Timeouts in Workflow Processes
Metalink Note 50468.1

Existing State of Packages Invalidated in PO Workflow
Metalink Note 303260.1.

Best Practices for Custom Order Entry Workflow Design
Metalink Note 402144.1

Embeded Scripts in 11.0
Metalink Note 183643.1

These part numbers cannot be searched from Metalink, but they can be linked from the Workflow Product Documentation Page Metalink Note 67183.1

Oracle Workflow Guide, Release 2.5. (2000) Part Number A75397-01

Oracle Workflow Developers Guide, Release 2.6.4 (June 2005) Part Number B15853-01

Oracle Workflow Administrators Guide, Release 2.6.3 (Sept 2003) Part Number B10283-02

Oracle Workflow API Reference, Release 2.6.3 (Sept 2003) Part Number B10286-02

 © Copyright 2008 by Daniel Stober