Value-Added Reseller (VAR) Language

(i) the software component known as ACUMATE developed and licensed by Lucent Technologies Inc. of Murray Hill, New Jersey, to Oracle and imbedded in the Oracle Retail Predictive Application Server – Enterprise Engine, Oracle Retail Category Management, Oracle Retail Item Planning, Oracle Retail Merchandise Financial Planning, Oracle Retail Advanced Inventory Planning and Oracle Retail Demand Forecasting applications.

(ii) the MicroStrategy Components developed and licensed by MicroStrategy Services Corporation (MicroStrategy) of McLean, Virginia to Oracle and imbedded in the MicroStrategy for Oracle Retail Data Warehouse and MicroStrategy for Oracle Retail Planning & Optimization applications.

(iii) the SeeBeyond component developed and licensed by Sun MicroSystems, Inc. (Sun) of Santa Clara, California, to Oracle and imbedded in the Oracle Retail Integration Bus application.

(iv) the Wavelink component developed and licensed by Wavelink Corporation (Wavelink) of Kirkland, Washington, to Oracle and imbedded in Oracle Retail Store Inventory Management.

(v) the software component known as Crystal Enterprise Professional and/or Crystal Reports Professional licensed by Business Objects Software Limited (“Business Objects”) and imbedded in Oracle Retail Store Inventory Management.

(vi) the software component known as Access Via™ licensed by Access Via of Seattle, Washington, and imbedded in Oracle Retail Signs and Oracle Retail Labels and Tags.

(vii) the software component known as Adobe Flex™ licensed by Adobe Systems Incorporated of San Jose, California, and imbedded in Oracle Retail Promotion Planning & Optimization application.

(viii) the software component known as Style Report™ developed and licensed by InetSoft Technology Corp. of Piscataway, New Jersey, to Oracle and imbedded in the Oracle Retail Value Chain Collaboration application.

(ix) the software component known as i-net Crystal-Clear™ developed and licensed by I-NET Software Inc. of Berlin, Germany, to Oracle and imbedded in the Oracle Retail Central Office and Oracle Retail Back Office applications.

(x) the software component known as WebLogic™ developed and licensed by BEA Systems, Inc. of San Jose, California, to Oracle and imbedded in the Oracle Retail Value Chain Collaboration application.

(xi) the software component known as DataBeacon™ developed and licensed by Cognos Incorporated of Ottawa, Ontario, Canada, to Oracle and imbedded in the Oracle Retail Value Chain Collaboration application.
# Contents

**Preface** ........................................................................................................................................ ix  
Audience ........................................................................................................................................ ix  
Related Documents.................................................................................................................. ix  
Customer Support .................................................................................................................. ix  
Review Patch Documentation ................................................................................................. x  
Oracle Retail Documentation on the Oracle Technology Network............................................ x  
Conventions.................................................................................................................................. x  

## 1 Pro*C Restart and Recovery .............................................................................. 1  
Table Descriptions and Definitions ..................................................................................... 1  
  restart_control ................................................................................................................... 2  
  restart_program_status ............................................................................................... 3  
  restart_program_history .......................................................................................... 4  
  restart_bookmark ...................................................................................................... 5  
  v_restart_x ................................................................................................................... 6  
Data Model Discussion ........................................................................................................ 6  
  Why restart_program_status and restart_bookmark are Separate Tables.... 6  
Physical Set-Up .................................................................................................................. 6  
Table and File-Based Restart/Recovery.............................................................................. 7  
API Functional Descriptions............................................................................................. 9  
  restart_init .................................................................................................................. 9  
  restart_file_init ......................................................................................................... 10  
  restart_commit .......................................................................................................... 10  
  restart_file_commit ................................................................................................. 10  
  restart_close ............................................................................................................. 11  
  parse_array_args ..................................................................................................... 11  
  restart_file_write ..................................................................................................... 11  
  restart_cat .................................................................................................................. 11  
  Restart Headers and Libraries.................................................................................... 12  
  Updated Restart Headers and Libraries .................................................................. 13  
  New Restart/Recovery Functions ............................................................................. 14  
Query-Based Commit Thresholds .................................................................................. 16  

## 2 Pro*C Multi-Threading ................................................................................... 17  
Threading Description ..................................................................................................... 17  
Threading Function for Query-Based ........................................................................ 18  
Restart View for Query-Based ................................................................................... 18  
Thread Scheme Maintenance ....................................................................................... 20  
  File-Based ............................................................................................................... 20  
  Query-Based ........................................................................................................... 21  
Batch Maintenance ....................................................................................................... 21
Preface

This operations guide serves as an Oracle Retail Merchandising System (RMS) solution reference to explain ‘backend’ processes and configuration.

Audience

Anyone who has an interest in better understanding the inner workings of the Retail Merchandising System can find valuable information in this guide. There are three audiences in general for whom this guide is written:

- System analysts and system operation personnel:
  - Who are looking for information about Retail Merchandising System processes internally or in relation to the systems across the enterprise
  - Who operate the Retail Merchandising System on a regular basis
- Integrators and implementation staff who have the overall responsibility for implementing the Retail Merchandising System in their enterprise
- Business analysts who are looking for information about processes and interfaces to validate the support for business scenarios within the Retail Merchandising System and other systems across the enterprise

Related Documents

You can find more information about this product in these resources:

- Oracle Retail Merchandising System Installation Guide
- Oracle Retail Merchandising System Release Notes
- Oracle Retail Merchandising System Data Model
- Oracle Retail Merchandising System Online Help
- Oracle Retail Merchandising System User Guide
- Oracle Retail Merchandising System Operations Guide (Volumes 1 and 2)
- Oracle Retail Sales Audit User Guide
- Oracle Retail Trade Management User Guide
- Oracle Retail Merchandising Batch Schedule
- Oracle Retail Merchandising Data Conversion Operations Guide
- Oracle Retail Merchandising Implementation Guide

Customer Support

https://metalink.oracle.com

When contacting Customer Support, please provide the following:

- Product version and program/module name
- Functional and technical description of the problem (include business impact)
- Detailed step-by-step instructions to re-create
- Exact error message received
- Screen shots of each step you take
Review Patch Documentation

For a base release (".0" release, such as 13.0), Oracle Retail strongly recommends that you read all patch documentation before you begin installation procedures. Patch documentation can contain critical information related to the base release, based on new information and code changes that have been made since the base release.

Oracle Retail Documentation on the Oracle Technology Network

In addition to being packaged with each product release (on the base or patch level), all Oracle Retail documentation is available on the following Web site:

http://www.oracle.com/technology/documentation/oracle_retail.html

Documentation should be available on this Web site within a month after a product release. Note that documentation is always available with the packaged code on the release date.

Conventions

Navigate: This is a navigate statement. It tells you how to get to the start of the procedure and ends with a screen shot of the starting point and the statement “the Window Name window opens.”

Note: This is a note. It is used to call out information that is important, but not necessarily part of the procedure.

This is a code sample

It is used to display examples of code

A hyperlink appears like this.
Pro*C Restart and Recovery

RMS has implemented a restart recovery process in most of its batch architecture. The general purpose of restart/recovery is to:

- Recover a halted process from the point of failure
- Prevent system halts due to large numbers of transactions
- Allow multiple instances of a given process to be active at the same time

Further, the RMS restart/recovery tracks batch execution statistics and does not require DBA authority to execute.

The restart capabilities revolve around a program’s logical unit of work (LUW). A batch program processes transactions, and commit points are enabled based on the LUW. LUWs consist of a relatively unique transaction key (such as sku/store) and a maximum commit counter. Commit events take place after the number of processed transaction keys meets or exceeds the maximum commit counter. For example, every 10,000 sku/store combinations, a commit occurs. At the time of the commit, key data information that is necessary for restart is stored in the restart tables. In the event of a handled or un-handled exception, transactions will be rolled back to the last commit point, and upon restart the key information will be retrieved from the tables so that processing can continue from the last commit point.

Table Descriptions and Definitions

The RMS restart/recovery process is driven by a set of four tables. Refer to the diagram for the entity relationship diagram, followed by table descriptions.

**Table Descriptions**

- **restart control**
  - (PK) program_name
  - program_desc
  - driver_name
  - num_threads
  - update_allowed
  - process_flag
  - commit_max_ctr

- **restart program history**
  - restart_name
  - thread_val
  - start_time
  - program_name
  - num_threads
  - commit_max_ctr
  - restart_time
  - finish_time
  - shadow_pid
  - success_flag
  - non_fatal_err_flag
  - num_commits
  - avg_time btwn_commits

- **restart program status**
  - (PK) restart_name
  - (PK) thread_val
  - start_time
  - program_name
  - program_status
  - restart_flag
  - restart_time
  - finish_time
  - current_pid
  - current_operator_id
  - err_message
  - current_oracle_sid
  - current_shadow_pid

- **restart bookmark**
  - restart_name
  - thread_val
  - bookmark_string
  - application_image
  - out_file_string
  - non_fatal_err_flag
  - num_commits
  - avg_time btwn_commits

**Entity Relationship**
### restart_control

The restart_control table is the master table in the restart/recovery table set. One record exists on this table for each batch program that is run with restart/recovery logic in place. The restart/recovery process uses this table to determine:

- Whether the restart/recovery is table-based or file-based
- The total number of threads used for each batch program
- The maximum records that will be processed before a commit event takes place
- The driver for the threading (multi-processing) logic.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>program_name</td>
<td>varchar</td>
<td>Batch program name</td>
</tr>
<tr>
<td>program_desc</td>
<td>varchar</td>
<td>A brief description of the program function</td>
</tr>
<tr>
<td>driver_name</td>
<td>varchar</td>
<td>Driver on query, for example, department (non-updatable)</td>
</tr>
<tr>
<td>num_threads</td>
<td>num</td>
<td>Number of threads used for current process</td>
</tr>
<tr>
<td>update_allowed</td>
<td>varchar</td>
<td>Indicates whether user can update thread numbers or if done programatically</td>
</tr>
<tr>
<td>process_flag</td>
<td>varchar</td>
<td>Indicates whether process is table-based (T) or file-based (F)</td>
</tr>
<tr>
<td>commit_max_ctr</td>
<td>num</td>
<td>Numeric maximum value for counter before commit occurs</td>
</tr>
</tbody>
</table>
### restart_program_status

The restart_program_status table is the table that holds record keeping information about current program processes. The number of rows for a program on the status table will be equal to its num_threads value on the restart_control table. The status table is modified during restart/recovery initialization and close logic. For table-based processing, the restart/recovery initialization logic will assign the next available thread to a program based on the program status and restart flag. For file-based processing, the thread value is passed in from the input file name. Once a thread has been assigned the program_status is updated to prevent the assignment of that thread to another process. Information will be logged on the current status of a given thread, as well as record keeping information such as operator and process timing information.

**Setup Note:** Allow row level locking and ‘dirty reads’ (do not wait for rows to be unlocked for table read).

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>restart_name</td>
<td>varchar2 50</td>
<td>Program name</td>
</tr>
<tr>
<td>thread_val</td>
<td>num 10</td>
<td>Thread counter</td>
</tr>
<tr>
<td>start_time</td>
<td>date dd-mon-yy hh:mi:ss</td>
<td>Initiation time for the program</td>
</tr>
<tr>
<td>program_name</td>
<td>varchar2 25</td>
<td>Program name</td>
</tr>
<tr>
<td>program_status</td>
<td>varchar2 25</td>
<td>Started, aborted, aborted in init, aborted in process, aborted in final, completed, ready for start</td>
</tr>
<tr>
<td>restart_flag</td>
<td>varchar2 1</td>
<td>Automatically set to ‘N’ after abnormal end, must be manually set to ‘Y’ for program to restart</td>
</tr>
<tr>
<td>restart_time</td>
<td>date dd-mon-yy hh:mi:ss</td>
<td>Restart time for the program</td>
</tr>
<tr>
<td>finish_time</td>
<td>date dd-mon-yy hh:mi:ss</td>
<td>Completion time for the program</td>
</tr>
<tr>
<td>current_pid</td>
<td>num 15</td>
<td>Starting program id</td>
</tr>
<tr>
<td>current_operator_id</td>
<td>varchar2 20</td>
<td>Operator that started the program</td>
</tr>
<tr>
<td>err_message</td>
<td>varchar2 255</td>
<td>Record that caused program abort &amp; associated error message</td>
</tr>
<tr>
<td>current_oracle_sid</td>
<td>num 15</td>
<td>Oracle SID for the session associated with the current process</td>
</tr>
<tr>
<td>current_shadow_pid</td>
<td>num 15</td>
<td>O/S process ID for the shadow process associated with the current process. It is used to locate the session trace file when a process is not finished successfully.</td>
</tr>
</tbody>
</table>
**restart_program_history**

The restart_program_history table will contain one record for every successfully completed program thread with restart/recovery logic. Upon the successful completion of a program thread, its record on the restart_program_status table will be inserted into the history table. Table purgings will be at user discretion.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Length/Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>restart_name</td>
<td>varchar2</td>
<td>50</td>
<td>Program name</td>
</tr>
<tr>
<td>thread_val</td>
<td>Num</td>
<td>10</td>
<td>Thread counter</td>
</tr>
<tr>
<td>start_time</td>
<td>Date</td>
<td></td>
<td>dd-mon-yy hh:mi:ss</td>
</tr>
<tr>
<td>program_name</td>
<td>varchar2</td>
<td>25</td>
<td>Program name</td>
</tr>
<tr>
<td>num_threads</td>
<td>Num</td>
<td>10</td>
<td>Number of threads</td>
</tr>
<tr>
<td>commit_max_ctr</td>
<td>num</td>
<td>6</td>
<td>Numeric maximum value for counter before commit occurs</td>
</tr>
<tr>
<td>restart_time</td>
<td>date</td>
<td></td>
<td>dd-mon-yy hh:mi:ss</td>
</tr>
<tr>
<td>finish_time</td>
<td>date</td>
<td></td>
<td>dd-mon-yy hh:mi:ss</td>
</tr>
<tr>
<td>shadow_pid</td>
<td>num</td>
<td>15</td>
<td>O/S process ID for the shadow process associated with the process. It is used to locate the session trace file.</td>
</tr>
<tr>
<td>success_flag</td>
<td>varchar2</td>
<td>1</td>
<td>Indicates whether the process finished successfully (reserved for future use)</td>
</tr>
<tr>
<td>non_fatal_err_flag</td>
<td>varchar2</td>
<td>1</td>
<td>Indicates whether non-fatal errors have occurred for the process</td>
</tr>
<tr>
<td>num_commits</td>
<td>num</td>
<td>12</td>
<td>Total number of commits for the process. The possible last commit when restart/recovery is closed is not counted.</td>
</tr>
<tr>
<td>avg_time_btwn_commits</td>
<td>num</td>
<td>12</td>
<td>Accumulated average time between commits for the process. The possible last commit when restart/recovery is closed is not counted.</td>
</tr>
</tbody>
</table>
restart_bookmark

When a restart/recovery program thread is currently active, its state is started or aborted, and a record for it exists on the restart_bookmark table. Restart/recovery initialization logic inserts the record into the table for a program thread. The restart/recovery commit process updates the record with the following restart information:

- A concatenated string of key values for table processing
- A file pointer value for file processing
- Application context information such as counters and accumulators

The restart/recovery closing process will delete the program thread record if the program finishes successfully. In the event of a restart, the program thread information on this table will allow the process to begin from the last commit point.

<table>
<thead>
<tr>
<th>restart_bookmark</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(PK) restart_name</td>
<td>varchar2</td>
<td>50</td>
</tr>
<tr>
<td>(PK) thread_val</td>
<td>num</td>
<td>10</td>
</tr>
<tr>
<td>bookmark_string</td>
<td>varchar2</td>
<td>255</td>
</tr>
<tr>
<td>application_image</td>
<td>varchar2</td>
<td>1000</td>
</tr>
<tr>
<td>out_file_string</td>
<td>varchar2</td>
<td>255</td>
</tr>
<tr>
<td>non_fatal_err_flag</td>
<td>varchar2</td>
<td>1</td>
</tr>
<tr>
<td>num_commits</td>
<td>num</td>
<td>12</td>
</tr>
<tr>
<td>avg_time_btwn_commits</td>
<td>num</td>
<td>12</td>
</tr>
</tbody>
</table>
**v_restart_x**

Restart views will be used for query-based programs that require multi-threading. Separate views will be created for each threading driver, for example, department or store. A join will be made to a view based on threading driver to force the separation of discrete data into particular threads. Please see the threading discussion for more details.

<table>
<thead>
<tr>
<th>v_restart_x</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>driver_name</td>
<td>varchar2</td>
<td>example dept, store, region, etc.</td>
</tr>
<tr>
<td>num_threads</td>
<td>number</td>
<td>total number of threads in set (defined on restart control)</td>
</tr>
<tr>
<td>driver_value</td>
<td>number</td>
<td>will be the numeric value of the driver_name</td>
</tr>
<tr>
<td>thread_val</td>
<td>number</td>
<td>thread value defined for driver_value and num_threads combination</td>
</tr>
</tbody>
</table>

**Data Model Discussion**

**Why restart_program_status and restart_bookmark are Separate Tables**

The initialization process needs to fetch all of the rows associated with restart_name/schema, but will only update one row. The commit process will continually lock a row with a specific restart_name and thread_val. The data involved with these two processes is separated into two tables to reduce the number of hangs that could occur due to locked rows. Even if you allow ‘dirty reads’ on locked rows, a process will still hang if it attempts to do an update on a locked row. The commit process is only interested in a unique row, so if we move the commit process data to a separate table with row level (not page level) locking, there will not be contention issues during the commit. With the separate tables, the initialization process will now see fewer problems with contention because rows will only be locked twice, at the beginning and end of the process.

**Physical Set-Up**

The restart/recovery process needs to be as robust as possible in the event of database related failure. The costs outweigh the benefits of placing the restart/recovery tables in a separate database. The tables should, however, be set up in a separate, mirrored table space with a separate rollback segment.
Table and File-Based Restart/Recovery

The restart/recovery process works by storing all the data necessary to resume processing from the last commit point. Therefore, the necessary information will be updated on the restart_bookmark table before the processed data is committed. Query-based and file-based modules will store different information on the restart tables, and will therefore call different functions within the restart/recovery API to perform their tasks.

When a program’s process is query-based, that is, a module is driven by a driving query that processes the retrieved rows, then the information that is stored on the restart_bookmark table is related to the data retrieved in the driving query. If the program fails while processing, the information that is stored on the restart-tables can be used in the conditional where-clause of the driving query to only retrieve data that has yet to be processed since the last commit event.

File-based processing, however, simply needs to store the file location at the time of the last commit point. This file’s byte location is stored on the restart_bookmark table and will be retrieved at the time of a restart. This location information will be used to seek forward in the re-opened file to the point at which the data was last committed.

Because there is different information being saved to and retrieved from the restart_bookmark table for each of the different types of processing, different functions will need to be called to perform the restart/recovery logic. The query-based processing will call the restart_init or retek_init and restart_commit or retek_commit functions while the file-based processing will call the restart_file_init and restart_file_commit functions.

In addition to the differences in API function calls, the batch processing flow of the restart/recovery will differ between the files. Table-based restart/recovery will need to use a priming fetch logical flow, while the file-based processing will usually read lines in a batch. Table-based processing requires its structure to ensure that the LUW key has changed before a commit event can be allowed to occur, while the file-based processing does not need to evaluate the LUW, which can typically be thought of as the type of transaction being processed by the input file.

The following diagram depicts table-based Restart/Recovery program flow:
Table-Based Restart/Recovery Program Flow

The following diagram depicts file-based Restart/Recovery program flow.
Initialization logic:
- Variable declarations
- File initialization
- Call restart_init() or restart_file_init() function - will determine start or restart logic
- First fetch on driving query

Start logic: initialize counters/accumulators to start values

Restart logic:
- Parse application_image field on bookmark table into counters/accumulators
- Initialize counters/accumulators to values of parsed fields

Process/commit loop:
- Process updates and manipulations
- Fetch new record
- Create varchar from counters/accumulators to pass into application_image field on restart_bookmark table
- Call restart_commit() or restart_file_commit()

Close logic:
- Reset pointers
- Close files/curators
- Call restart_close()

API Functional Descriptions

restart_init

An initialization function for table-based batch processing.
The process gathers information from the restart control tables
- Total number of threads for a program and thread value assigned to current process.
- Number of records to loop through in driving cursor before commit (LUW).
- Start string - bookmark of last commit to be used for restart or a null string if current process is an initial start and initializes the restart record-keeping (restart_program_status).
- Program status is changed to ‘started’ for the first available thread.
- Operational information is updated: operator, process, start_time, etc. and bookmarking (restart_bookmark) tables.
- On an initial start, a record is inserted.
- On restart, the start string and application context information from the last commit is retrieved.
restart_file_init

An initialization function for file-based batch processing. It is called from program modules.

1. The process gathers information from the restart control tables:
   - number of records to read from file for array processing and for commit cycle
   - file start point- bookmark of last commit to be used for restart or 0 for initial start

2. The process initializes the restart record-keeping (restart_program_status):
   - program status is changed to ‘started’ for the current thread
   - operational information is updated: operator, process, start_time, etc.

3. The process initializes the restart bookmarking (restart_bookmark) tables:
   - on an initial start, a record is inserted
   - on restart, the file starting point information and application context information from the last commit is retrieved

restart_commit

A function that commits the processed transaction for a given number of driving query fetches. It is called from program modules.

The process updates the restart_bookmark start string and application image information if a commit event has taken place:

- the current number of driving query fetches is greater than or equal to the maximum set in the restart_program_status table (and fetched in the restart_init function)
- the bookmark string of the last processed record is greater than or equal to the maximum set in the restart_program_status table (and fetched in the restart_init function)
- the bookmark string increments the counter
- the bookmark string sets the current string to be the most recently fetched key string

restart_file_commit

A function that commits processed transactions after reading a number of lines from a flat file. It is called from program modules.

The process updates the restart_bookmark table:

- start_string is set to the file pointer location in the current read of the flat file
- application image is updated with context information
**restart_close**

A function that updates the restart tables after program completion. The process determines whether the program was successful. If the program finished successfully:
- the `restart_program_status` table is updated with finish information and the status is reset
- the corresponding record in the `restart_bookmark` table is deleted
- the `restart_program_history` table has a copy of the `restart_program_status` table record inserted into it
- the `restart_program_status` is re-initialized

If the program ends with errors
- the transactions are rolled back
- the `program_status` column on the `restart_program_status` table is set to ‘aborted in *’ where * is one of the three main functions in batch: init, process or final
- the changes are committed

**parse_array_args**

This function parses a string into components and places results into multidimensional array. It is only called within API functions and will never be called in program modules.

The process is passed a string to parse and a pointer to an array of characters. The first character of the passed string is the delimiter.

**restart_file_write**

This function will append output in temporary files to final output files when a commit point is reached. It is called from program modules.

**restart_cat**

This function contains the logic that appends one file to another. It is only called within the restart/recovery API functions and will never be called directly in program modules.
Restart Headers and Libraries

The restart.h and the std_err.h header files are included in retek.h to utilize the restart/recovery functionality.

restart.h
This library header file contains constant, macro substitutions, and external global variable definitions as well as restart/recovery function prototypes.

The global variables that are defined include:
- the thread number assigned to the current process
- the value of the current process’s thread maximum counter
  - for table-based processing, it is equal to the number of iterations of the driving query before a commit can take place
  - for file-based processing, it is equal to the number of lines that will be read from a flat file and processed using a structured array before a commit can take place
- the current count of driving query iterations used for table-based processing or the current array index used in file-based processing
- the name assigned to the program/logical unit of work by the programmer. It is the same as the restart_name column on the restart_program_status, restart_program_history, and restart_bookmark tables

std_rest.h
This library header file contains standard restart variable declarations that are used visible in program modules.

The variable definitions that are included are:
- the concatenated string value of the fetched driving query key that is currently being processed
- the concatenated string value of the fetched driving query key that is next to be processed
- the error message passed to the restart_close function and updated to restart_program_status
- concatenated string of application context information, for example, counters & accumulators
- the name of the threading driver, for example, department, store, warehouse, etc.
- the total number of threads used by this program
- the pointer to pass to initialization function to retail number of threads value
Updated Restart Headers and Libraries

Restart/recovery performs the following, among other capabilities:

- Organizes global variables associated with restart recovery
- Allows the batch developer full control of restart recovery variables parameter passing during initialization
- Removes temporary write files to speed up the commit process
- Moves more information and processing from the batch code into the library code
- Adds more information into the restart recovery tables for tuning purposes

retek_2.h

This library header file is included by all C code within Retek and serves to centralize system includes, macro defines, globals, function prototypes, and, especially, structs for use in the new restart/recovery library.

The globals used by the old restart/recovery library are all discarded. Instead, each batch program declares variables needed and calls retek_init() to get them populated from restart/recovery tables. Therefore, only the following variables are declared:

- gi_no_commit: flag for NO_COMMIT command line option (used for tuning purposes)
- gi_error_flag: fatal error flag
- gi_non_fatal_err_flag: non-fatal error flag

In addition, a rtk_file struct is defined to handle all file interfaces associated with restart/recovery. Operation functions on the file struct are also defined.

```c
#define NOT_PAD            1000  /* Flag not to pad thread_val */
#define PAD                1001  /* Flag to pad thread_val at the end */
#define TEMPLATE           1002  /* Flag to pad thread_val using filename template */
#define MAX_FILENAME_LEN   50

typedef struct
{
    FILE* fp;                /* File pointer */
    char  filename[MAX_FILENAME_LEN + 1]; /* Filename */
    int   pad_flag;         /* Flag whether to pad thread_val to filename */
} rtk_file;

int   set_filename(rtk_file* file_struct, char* file_name, int pad_flag);
FILE* get_FILE(rtk_file* file_struct);
int   rtk_print(rtk_file* file_struct, char* format, ...);
int   rtk_seek(rtk_file* file_struct, long offset, int whence);
```

The parameters retek_init() needs to populate are required to be passed in using a format known to retek_init(). A struct is defined here for this purpose. An array of parameters of this struct type is needed at each batch program. Other requirements are:
Need to be initialized at each batch program.

- The lengths of name, type and sub_type should not exceed the definitions here.
- Type can only be: "int", "uint", "long", "string", or "rtk_file".
- For type "int", "uint" or "long", use "" as sub_type.
- For type "string", sub_type can only be "S" (start string) unless the string is the thread
  value or number of threads, in which case use "" as sub_type or "I" (image string).
- For type "rtk_file", sub_type can only be "I" (input) or "O" (output).

#define NULL_PARA_NAME        51
#define NULL_PARA_TYPE        21
#define NULL_PARA_SUB_TYPE    2
typedef struct
{
    char name[NULL_PARA_NAME];
    char type[NULL_PARA_TYPE];
    char sub_type[NULL_PARA_SUB_TYPE];
} init_parameter;

New Restart/Recovery Functions

Starting from release 9.0, all new batch programs are coded using the new
restart/recovery functions. Batch programs using the old restart/recovery API functions
are still in use. Therefore, Oracle Retail is currently maintaining two sets of
restart/recovery libraries.

int retek_init(int num_args, init_parameter *parameter, ...)

retek_init initializes restart/recovery (for both table- and file-based):

1. Pass in num_args as the number of elements in the init_parameter array, then the
   init_parameter array, then variables a batch program needs to initialize in the order
   and types defined in the init_parameter array. Note that all int, uint and long
   variables need to be passes by reference.
2. Get all global and module level values from databases.
3. Initialize records for RESTART_PROGRAM_STATUS and RESTART_BOOKMARK.
4. Parse out user-specified initialization variables (variable arg list).
5. Return NO_THREAD_AVAILABLE if no qualified record in RESTART_CONTROL
   or RESTART_PROGRAM_STATUS.
6. Commit work.
int retek_commit(int num_args, ...)  
retek_commit checks and commits if needed (for both table- and file-based):  
1. Pass in num_args, then variables for start_string first, and those for image string (if needed) second. The num_args is the total number of these two groups. All are string variables and are passed in the same order as in retek_init();  
2. Concatenate start_string either from passed in variables (table-based) or from ftell of input file pointers (file-based);  
3. Check if commit point reached (counter check and, if table-based, start string comparison);  
4. If reached, concatenated image_string from passed in variables (if needed) and call internal_commit() to get out_file_string and update RESTART_BOOKMARK;  
5. If table-based, increment pl_current_count and update ps_cur_string.

int commit_point_reached(int num_args, ...)  
commit_point_reached checks if the commit point has been reached (for both table- and file-based). The difference between this function and the check in retek_commit() is that here the pl_current_count and ps_cur_string are not updated. This checking function is designed to be used with retek_force_commit(), and the logic to ensure integrity of LUW exists in user batch program. It can also be used together with retek_commit() for extra processing at the time of commit.  
1. Pass in num_args, then all string variables for start_string in the same order as in retek_init(). The num_args is the number of variables for start_string. If no start_string (as in file-based), pass in NULL.  
2. For table-based, if pl_current_count reaches pl_max_counter and if newly concatenated bookmark string is different from ps_cur_string, return 1; otherwise return 0.  
3. For file-based, if pl_current_count reaches pl_max_counter return 1; otherwise return 0.

int retek_force_commit(int num_args, ...)  
retek_force_commit always commits (for both table- and file-based):  
1. Pass in num_args, then variables for start_string first, and those for image string (if needed) second. The num_args is the total number of these two groups. All are string variables and are passed in the same order as in retek_init().  
2. Concatenate start_string either from passed in variables (table-based) or from ftell of input file pointers (file-based).  
3. Concatenated image_string from passed in variables (if needed) and call internal_commit() to get out_file_string and update RESTART_BOOKMARK.  
4. If table-based, increment pl_current_count and update ps_cur_string.
int retek_close(void)
retek_close closes restart/recovery (for both table- and file-based):
1. If gi_error_flag or NO_COMMIT command line option is TRUE, rollback all database changes.
2. Update RESTART_PROGRAM_STATUS according to gi_error_flag.
3. If no gi_error_flag, insert record into RESTART_PROGRAM_HISTORY with information fetched from RESTART_CONTROL, RESTART_PROGRAM_BOOKMARK and RESTART_PROGRAM_STATUS tables.
4. If no gi_error_flag, delete RESTART_BOOKMARK record.
5. Commit work.
6. Close all opened file streams.

Int retek_refresh_thread(void)
Refreshes a program’s thread so that it can be run again.
1. Updates the RESTART_PROGRAM_STATUS record for the current program’s PROGRAM_STATUS to be ‘ready for start’.
2. Deletes any RESTART_BOOKMARK records for the current program.
3. Commits work.

void increment_current_count(void)
increment_current_count increases pl_current_count by 1.

Note: This is called from get_record() of intrface.pc for file-based I/O.

int parse_name_for_thread_val(char* name)
parse_name_for_thread_val parses thread value from the extension of the specified file name.

int is_new_start(void)
is_new_start checks if current run is a new start; if yes, return 1; otherwise 0.

Query-Based Commit Thresholds
The restart capabilities revolve around a program’s logical unit of work (LUW). A batch program processes transactions and enables commit points based on the LUW. An LUW is comprised of a transaction key (such as item-store) and a maximum commit counter. Commit events occur after a given number of transaction keys are processed. At the time of the commit, key data information that is necessary for restart is stored in the restart table. In the event of a handled or un-handled exception, transactions will be rolled back to the last commit point. Upon restart the restart key information will be retrieved from the tables so that processing can resume with the unprocessed data.
Pro*C Multi-Threading

Processing multiple instances of a given program can be accomplished through “threading”. This requires driving cursors to be separated into discrete segments of data to be processed by different threads. This will be accomplished through stored procedures that will separate threading mechanisms (for example, departments or stores) into particular threads given value (for example, department 1001) and the total number of threads for a given process.

File-based processing will not truly “thread” its processing. The same data file will never be acted upon by multiple processes. Multi-threading will be accomplished by dividing the data into separate files each of which will be acted upon by a separate process. The thread value is related to the input file. This is necessary to ensure that the appropriate information can be tied back to the relevant file in the event of a restart.

RMS has a store length of ten digits. Therefore, thread values, which can be based upon the store number, should allow ten digits as well. Due to the thread values being declared as ‘C’ variables of type int (long), the system is restricting thread values to nine digits.

This does not mean that you cannot use ten digit store numbers. It means that if you do use ten digit store numbers you cannot use them as thread values.

Threading Description

The use of multiple threads or processes in Oracle Retail batch processing will increase efficiency and decrease processing time. The design of the threading process has allowed maximum flexibility to the end user in defining the number of processes over which a program should be divided.

Originally, the threading function was going to be used directly in the driving queries. This was found, however, to be unacceptably slow. Instead of using the function call directly in the driving queries, the designs call for joining driving query tables to a view (for example, v_restart_store) that includes the function.
Threading Function for Query-Based

A stored procedure has been created to determine thread values. Restart_thread_return returns a thread value derived from a numeric driver value, such as department number, and the total number of threads in a given process. Retailers should be able to determine the best algorithm for their design, and if a different means of segmenting data is required, then either the restart_thread_return function can be altered, or a different function can be used in any of the views in which the function is contained.

Currently the restart_thread_return function is a very simple modulus routine:

```sql
CREATE OR REPLACE FUNCTION RESTART_THREAD_RETURN(in_unit_value  NUMBER,
in_total_threads NUMBER)
RETURN NUMBER IS
  ret_val NUMBER;
BEGIN
  ret_val := MOD(ABS(in_unit_value), in_total_threads) + 1;
  RETURN ret_val;
END;
```

Restart View for Query-Based

Each restart view will have four elements:

- the name of the threading mechanism, driver_name
- the total number of threads in a grouping, num_threads
- the value of the driving mechanism, driver_value
- the thread value for that given combination of driver_name, num_threads, and driver_value, thread_val

The view will be based on the restart_control table and an information table such as DEPS or STORES. A row will exist in the view for every driver value and every total number of threads value. Therefore, if a retailer were to always use the same number of threads for a given driver (dept, store, etc.), then the view would be relatively small. As an example, if all of a retailer’s programs threaded by department have a total of 5 threads, then the view will contain only one value for each department. For example, if there are 10 total departments, 10 rows will exist in v_restart_dept. However, if the retailer wants to have one of the programs to have ten threads, then there will be 2 rows for every department: one for five total threads and one for ten total threads (for example, if 10 total departments, 20 rows will exist in v_restart_dept). Obviously, retailers should be advised to keep the number of total thread values for a thread driver to a minimum to reduce the scope of the table join of the driving cursor with the view.

Below is an example of how the same driver value can result in differing thread values. This example uses the restart_thread_return function as it currently is written to derive thread values.
<table>
<thead>
<tr>
<th>Driver_name</th>
<th>num_threads</th>
<th>driver_val</th>
<th>thread_val</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEPT</td>
<td>1</td>
<td>101</td>
<td>1</td>
</tr>
<tr>
<td>DEPT</td>
<td>2</td>
<td>101</td>
<td>2</td>
</tr>
<tr>
<td>DEPT</td>
<td>3</td>
<td>101</td>
<td>3</td>
</tr>
<tr>
<td>DEPT</td>
<td>4</td>
<td>101</td>
<td>2</td>
</tr>
<tr>
<td>DEPT</td>
<td>5</td>
<td>101</td>
<td>2</td>
</tr>
<tr>
<td>DEPT</td>
<td>6</td>
<td>101</td>
<td>6</td>
</tr>
<tr>
<td>DEPT</td>
<td>7</td>
<td>101</td>
<td>4</td>
</tr>
</tbody>
</table>

Below is an example of what a distribution of stores might look like given 10 stores and 5 total threads:

<table>
<thead>
<tr>
<th>Driver_name</th>
<th>num_threads</th>
<th>driver_val</th>
<th>thread_val</th>
</tr>
</thead>
<tbody>
<tr>
<td>STORE</td>
<td>5</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>STORE</td>
<td>5</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>STORE</td>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>STORE</td>
<td>5</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>STORE</td>
<td>5</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>STORE</td>
<td>5</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>STORE</td>
<td>5</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>STORE</td>
<td>5</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>STORE</td>
<td>5</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>STORE</td>
<td>5</td>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>
Thread Scheme Maintenance

View syntax:
The following is an example of the syntax needed to create the view for the multi-threading join, created with script (see threading discussion for details on restart_thread_return function):

```sql
create or replace view v_restart_store as
select rc.driver_name driver_name,
    rc.num_threads num_threads,
    s.store driver_value,
    restart_thread_return(s.store, rc.num_threads) thread_val
from restart_control rc, store s
where rc.driver_name = 'STORE'
```

There is a different threading scheme used within Oracle Retail Sales Audit (ReSA). Because ReSA needs to run 24 hours a day and seven days a week, there is no batch window. This means that there may be batch programs running at the same time that there are online users. ReSA solved this concurrency problem by creating a locking mechanism for data that is organized by store days. These locks provide a natural threading scheme. Programs that cycle through all of the store day data attempt to lock the store day first. If the lock fails, the program simply goes on to the next store day. This has the affect of automatically balancing the workload between all of the programs executing.

Thread Scheme Maintenance

All program names will be stored on the restart_control table along with a functional description, the query driver (dept, store, class, etc.) and the user-defined number of threads associated with them. Users should be able to scroll through all programs to view the name, description, and query driver, and if the update_allowed flag is set to true, to modify the number of threads (update is set to true).

File-Based

File based processing does not truly “multi-thread” and therefore the number of threads defined on restart_control will always be one. However, a restart_program_status record will need to be created for each input file that is to be processed for the program module. Further, the thread value that is assigned should be part of the input file name. The restart_parse_name function that is included in the program module will parse the thread value from the program name and use that to determine the availability and restart requirements on the restart_program_status table.

Refer to the beginning of this multi-threading section for a discussion of limits on using large (greater than nine digits) thread values.
Query-Based

When the number of threads is modified in the restart_control table, the form should first validate that no records for that program are currently being processed in the restart_program_status_table (that is, all records = ‘Completed’). The program should insert or delete rows depending on whether the new thread number is greater than or less than the old thread number. In the event that the new number is less than the previous number, all records for that program_name with a thread number greater than the new thread number will be deleted. If the new number is greater than the old number, new rows will be inserted. A new record will be inserted for each restart_name/thread_val combination.

For example if the batch program SALDLY has its number of processes changed from 2 to 3, then an additional row (3) will be added to the restart_program_status_table. Likewise, if the number of threads was reduced to 1 in this example, rows 2 and 3 would be deleted.

Original restart_program_status table:

<table>
<thead>
<tr>
<th>row #</th>
<th>restart_name</th>
<th>thread_val</th>
<th>program_name etc…</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SALDLY</td>
<td>1</td>
<td>SALDLY</td>
</tr>
<tr>
<td>2</td>
<td>SALDLY</td>
<td>2</td>
<td>SALDLY</td>
</tr>
</tbody>
</table>

restart_program_status table after insert:

<table>
<thead>
<tr>
<th>row #</th>
<th>restart_name</th>
<th>thread_val</th>
<th>program_name etc…</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SALDLY</td>
<td>1</td>
<td>SALDLY</td>
</tr>
<tr>
<td>2</td>
<td>SALDLY</td>
<td>2</td>
<td>SALDLY</td>
</tr>
<tr>
<td>3</td>
<td>SALDLY</td>
<td>3</td>
<td>SALDLY</td>
</tr>
</tbody>
</table>

restart_program_status table after delete:

<table>
<thead>
<tr>
<th>row #</th>
<th>restart_name</th>
<th>thread_val</th>
<th>program_name etc…</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SALDLY</td>
<td>1</td>
<td>SALDLY</td>
</tr>
</tbody>
</table>

Users should also be able to modify the commit_max_ctr column in restart_program_status table. This will control the number of iterations in driving query or the number of lines read from a flat file that determine the logical unit of work (LUW).

Batch Maintenance

Users should be able to view the status of all records in restart_program_status table. This is where the user will come to view error messages from aborted programs, and statistics and histories of batch runs. The only fields that will be modifiable will be program_status and restart_flag. The user should be able to reset the restart_flag to ‘Y’ from ‘N’ on records with a status of aborted, started records to aborted in the event of an abend (abnormal termination), and all records in the event of a restore from tape/re-run of all batch.
Scheduling and Initialization of Restart Batch

Before any batch with restart/recovery logic is run, an initialization program should be run to update the status in the restart_program_status table. This program should update the program_status to ‘ready for start’ wherever a record’s program_status is ‘completed.’ This will leave unchanged all programs that ended unsuccessfully in the last batch run.

Pre- and Post-Processing

Due to the nature of the threading algorithm, individual programs might need a pre or a post program run to initialize variables or files before any of the threads have run or to update final data once all the threads are run. The decision was made to create pre-programs and post-programs in these cases rather than let the restart/recovery logic decide whether the currently processed thread is the first thread to start or the last thread to end for a given program.
Pro*C Array Processing

Oracle Retail batch architecture uses array processing to improve performance wherever possible. Instead of processing SQL statements using scalar data, data is grouped into arrays and used as bind variables in SQL statements. This improves performance by reducing the server/client and network traffic.

Array processing is used for select, insert, delete, and update statements. Oracle Retail typically does not statically define the array sizes, but uses the restart maximum commit variable as a sizing multiple. Users should keep this in mind when defining the system's maximum commit counters.

An important factor to keep in mind when using array processing is that Oracle does not allow a single array operation to be performed for more than 32000 records in one step. The Oracle Retail restart/recovery libraries have been updated to define macros for this value: MAX_ORACLE_ARRAY_SIZE.

All batch programs that use array processing need to limit the size of their array operations to MAX_ORACLE_ARRAY_SIZE.

If the commit max counter is used for array processing size, check it after the call to restart_init() and, if necessary, reset it to the maximum value if greater. If retek_init() is used to initialize, check the returned commit max counter and reset it to the maximum size if it is greater. In case of retek_init(), reset the library’s internal commit max counter by calling extern int limit_commit_max_ctr(unsigned int new_max_ctr).

If some other variable is used for sizing the array processing, the actual array-processing step will have to be encapsulated in a calling loop that performs the array operation in sub segments of the total array size where each sub-segment is at most MAX_ORACLE_ARRAY_SIZE large. Currently all Oracle Retail batch programs are implemented this way.
Pro*C Input and Output Formats

Oracle Retail batch processing will utilize input from both tables and flat files. Further, the outcome of processing can both modify data structures and write output data. Interfacing Oracle Retail with external systems is the main use of file based I/O.

General Interface Discussion

To simplify the interface requirements, Oracle Retail requires that all in-bound and out-bound file-based transactions adhere to standard file layouts. There are two types of file layouts, detail-only and master-detail, which are described below.

An interfacing API exists within Oracle Retail to simplify the coding and the maintenance of input files. The API provides functionality to read input from files, ensure file layout integrity, and write and maintain files for rejected transactions.

Standard File Layouts

The RMS interface library supports two standard file layouts; one for master/detail processing, and one for processing detail records only. True sub-details are not supported within the RMS base package interface library functions.

A 5-character identification code or record type identifies all records within an I/O file, regardless of file type. Valid record type values include the following:

- FHEAD—File Header
- FDETL—File Detail
- FTAIL—File Tail
- THEAD—Transaction Header
- TDETL—Transaction Detail
- TTAIL—Transaction Tail

Each line of the file must begin with the record type code followed by a 10-character record ID.

Detail Only Files

File layouts have a standard file header record, a detail record for each transaction to be processed, and a file trailer record. Valid record types are FHEAD, FDETL, and FTAIL.

Example:

FHEAD0000000001STKU1996010100000019960929
FDETL0000000002SKU10000004000000011011
FDETL0000000003SKU100000050003002001
FDETL0000000004SKU100000050003002001
FTAIL00000000050000000003
Master and Detail Files

File layouts will have a standard file header record, a set of records for each transaction to be processed, and a file trailer record. The transaction set will consist of a transaction set header record, a transaction set detail for detail within the transaction, and a transaction trailer record. Valid record types are FHEAD, THEAD, TDETL, TTAIL, and FTAIL.

Example:
FHEAD000000001RTV 19960908172000
THEAD00000000020000000000000119960909120200000000003R
TDETL000000000300000000000001SKU10000012
TTAIL0000000004000001
THEAD0000000005000000000000021996090912011215720131R
TDETL0000000006000000000000020000001UPC400100002667
TDETL000000000700000000000002000001UPC400100002643 0
TTAIL0000000008000002
FTAIL00000000090000000007

<table>
<thead>
<tr>
<th>Record Name</th>
<th>Field Name</th>
<th>Field Type</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>File Header</td>
<td>File Type</td>
<td>Char(5)</td>
<td>FHEAD</td>
<td>Identifies file record type</td>
</tr>
<tr>
<td></td>
<td>Record Descriptor</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>File Line Identifier</td>
<td>Number(10)</td>
<td>Specified by external system</td>
<td>Line number of the current file</td>
</tr>
<tr>
<td></td>
<td>File Type Definition</td>
<td>Char(4)</td>
<td>n/a</td>
<td>Identifies transaction type</td>
</tr>
<tr>
<td></td>
<td>File Create Date</td>
<td>Date</td>
<td>Create date</td>
<td>Date file was written by external system</td>
</tr>
<tr>
<td>Transaction Header</td>
<td>File Type Record Descriptor</td>
<td>Char(5)</td>
<td>THEAD</td>
<td>Identifies file record type</td>
</tr>
<tr>
<td></td>
<td>File Line Identifier</td>
<td>Number(10)</td>
<td>Specified by external system</td>
<td>Line number of the current file</td>
</tr>
<tr>
<td></td>
<td>Transaction Set Control Number</td>
<td>Char(14)</td>
<td>Specified by external system</td>
<td>Used to force unique transaction check</td>
</tr>
<tr>
<td></td>
<td>Transaction Date</td>
<td>Char(14)</td>
<td>Specified by external system</td>
<td>Date the transaction was created in external system</td>
</tr>
<tr>
<td>Transaction Detail</td>
<td>File Type Record Descriptor</td>
<td>Char(5)</td>
<td>TDETL</td>
<td>Identifies file record type</td>
</tr>
<tr>
<td></td>
<td>File Line Identifier</td>
<td>Number(10)</td>
<td>Specified by external system</td>
<td>Line number of the current file</td>
</tr>
<tr>
<td></td>
<td>Transaction Set Control Number</td>
<td>Char(14)</td>
<td>Specified by external system</td>
<td>Used to force unique transaction check</td>
</tr>
<tr>
<td>Record Name</td>
<td>Field Name</td>
<td>Field Type</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------------------</td>
<td>------------</td>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Detail Sequence</td>
<td>Number</td>
<td>Char(6)</td>
<td>Specified by external system</td>
<td>Sequential number assigned to detail records within a transaction</td>
</tr>
<tr>
<td>Transaction</td>
<td>Trailer File Type</td>
<td>Char(5)</td>
<td>TTAIL</td>
<td>Identifies file record type</td>
</tr>
<tr>
<td>Transaction</td>
<td>Trailer File Type</td>
<td>Number(10)</td>
<td>Specified by external system</td>
<td>Line number of the current file</td>
</tr>
<tr>
<td>Transaction</td>
<td>Detail Line Count</td>
<td>Number(6)</td>
<td>Sum of detail lines</td>
<td>Sum of the detail lines within a transaction</td>
</tr>
<tr>
<td>File Trailer</td>
<td>File Type Record</td>
<td>Char(5)</td>
<td>FTAIL</td>
<td>Identifies file record type</td>
</tr>
<tr>
<td>File Trailer</td>
<td>Descriptor</td>
<td>Number(10)</td>
<td>Specified by external system</td>
<td>Line number of the current file</td>
</tr>
<tr>
<td>Total Transaction</td>
<td>Line Count</td>
<td>Number(10)</td>
<td>Sum of all transaction lines</td>
<td>All lines in file less the file header and trailer records</td>
</tr>
</tbody>
</table>
Electronic Data Interchange (EDI)

Starting with release 7.0, EDI files used or created by RMS are in a generic format: RMS no longer supports particular EDI standards. By processing EDI output and input in a generic format, RMS is no longer limited to a single standard, which allows Oracle Retail customers to better utilize any and all standards they choose to use. Translating EDI input and output files into any format from any format by third-party software is an industry “best practice”.

Formerly, EDI transactions in RMS conformed to ASC X12/VICS (version 3040) and ANA/TRADACOMS standards. EDI transactions are now expected to be in a format that adheres to the RMS file interfacing standards. Both in-bound and out-bound files are written in a fixed field layout with standard file header and trailer records. Transaction information is included in master/detail or detail-only records. The layouts are consistent with interface files used elsewhere in the RMS.

RMS EDI batch processes write out-bound transaction files into the generic layout format, which are then translated by the third-party software into the standard required by each trading partner. The post-translated versions are transmitted to the trading partner. In-bound transactions should be formatted by the trading partner in a predetermined standard, transmitted, and then translated by the Oracle Retail retailer’s translation software into the generic file layout. The generic file is used as the input file for RMS EDI batch processing.

It is impractical for Oracle Retail to continue to maintain code that supports any particular EDI standard. There are multiple viable standards that are utilized by vendors and retailers. Further, those standards have multiple versions. Most retailers are already using software to map and translate EDI transactions into the required standard or version. There are excellent third-party software packages, such as Sterling Software’s Gentran™ translator, that effectively translate in-bound and out-bound transactions into the necessary formats. The use of third-party translation software is not only the common business practice, but also the best business practice of today’s retailer.
This chapter summarizes the configuration, architecture and features for many RETL programs utilized in RMS/ReSA extractions. These extractions were initially designed for Oracle Retail Data Warehouse (RDW) and can be used for some other application in the retailer’s enterprise.

For information about RMS RETL extractions for an application such as Advanced Inventory Planning (AIP), see the RMS 10.1.9 Addendum to the Operations Guide.

For more information about the RETL tool, see the latest RETL Programmer’s Guide.

Overview

RMS works in conjunction with the RETL framework. This architecture optimizes a high performance data processing tool that allows database batch processes to take advantage of parallel processing capabilities.

The RETL framework runs and parses through the valid operators composed in XML scripts.

Architectural Design

The diagrams below illustrate the extraction processing architecture for RMS and for ReSA. Instead of managing the change captures as they occur in the source system during the day, the process involves extracting the current data from the source system. The extracted data is output to flat files. These flat files are then available for consumption by a product such as RDW.

The target system, (RDW, for example), has its own way of completing the transformations and loading the necessary data into its system, where it can be used for further processing in the environment.
RMS Extraction Architecture

The architecture relies upon the use of well-defined flows specific to the RMS database. The resulting output is comprised of data files written in a well-defined schema file format. This extraction includes no destination specific code.

ReSA Extraction Architecture

The architecture relies upon the use of well-defined flows specific to ReSA input schema files. The resulting output is comprised of data files written in a well-defined schema file format. This extraction includes no destination specific code.
Configuration

RETL
Before trying to configure and run RMS ETL, install RETL version 12.0 or later, which is required to run RMS RETL. Run the ‘verify_retl’ script (included as part of the RETL installation) to ensure that RETL is working properly before proceeding.

RETL User and Permissions
RMS ETL is installed and run as the RETL user. Additionally, the permissions are set up as per the RETL Programmer’s Guide. RMS ETL reads data, creates, deletes, and updates tables. If these permissions are not set up properly, extractions fail.

Environment Variables
See the RETL Programmer’s Guide for RETL environment variables that must be set up for your version of RETL. You will need to set MMHOME to your base directory for RMS RETL. This is the top level directory that you selected during the installation process. In your .kshrc, you should add a line such as the following:

```
export MMHOME=<base directory for RMS ETL>/dwi12.0/dev
```

dwi_config.env Settings
Make sure to review the environmental parameters in the dwi_config.env file before executing batch modules. There are several variables you must change depending upon your local settings:

For example:

```
export DBNAME=int9i
export RMS_OWNER=steffej_rms1011
export BA_OWNER=rmsint1011
export ORACLE_PORT="1524"
export ORACLE_HOST="mspdev38"
export CONN_TYPE="thin"
```

You must set up the environment variable PASSWORD in dwi_config.env. In the example below, adding the line to the dwi_config.env causes the password ‘mypasswd’ to be used to log into the database:

```
export PASSWORD=mypasswd
```
Steps to Configure RETL

1. Log in to the UNIX server with a UNIX account that will run the RETL scripts.
2. Change directories to $MMHOME/rfx/etc.
3. Modify the dwi_config.env script:
   - DBNAME refers to the name of the RMS database.
   - RMS_OWNER refers to the username of the RMS schema owner.
   - BA_OWNER refers to the username of the RMSE batch user.
   - ORACLE_HOST refers to the database server name.
   - ORACLE_PORT refers to the database port number.
   - MAX_NUM_COLS refers to the maximum number of columns from which RETL selects records.
   - CONN_TYPE refers to the driver type used in RMS connection to the database.
     - CONN_TYPE=thin: Lighter weight, Java only driver.
     - CONN_TYPE=oci: Oracle Client Interface (OCI) driver. To be used for Real Application Clusters (RAC) implementations.
   - LOAD_TYPE refers to the SQL*Loader method used to load data to the database. Direct load can be used when you have a large amount of data to load quickly. A direct path load can quickly load and index large amounts of data. It can also load data into either an empty or non-empty table. Conventional, not direct load, is the Oracle and RMS default. There are restrictions on when direct load can be used. Before using direct load, clients should research direct vs conventional load, and the restrictions to using direct load.
     - LOAD_TYPE=conventional: loads the data using the conventional SQL-loader method for Oracle.
     - LOAD_TYPE=direct: loads the data using the direct SQL_loader method for Oracle.
   - IM_PROCESS refers to the method used to process item dimensional data. A number of factors determine which method performs best, including the volume of items a client has and how often changes are made to items. The higher the ratio of item changes to total items, the more likely one would benefit from the full snapshot method. Clients should test the methods to see what performs best for them. The default value for this setting is full snapshot.
     - IM_PROCESS=D: process the data using delta mode, which will process item data based on daily changed data only.
     - IM_PROCESS=F: process the data using snapshot mode, which will process items based on a full snapshot of all data.
   - ISCL_PROCESS refers to the method used to process item supplier location dimension data. A number of factors determine which method performs best, including the volume of item location and supplier combinations a client has and how often changes are made to those. The higher the ratio of changes to the combinations, the more likely one would benefit from the full snapshot method. Clients should test the methods to see what performs best for them. The default value for this setting is full snapshot.
     - ISCL_PROCESS=D: process the data using delta mode, which will process item supplier location data based on daily changed data only.
Program Features

- ISCL_PROCESS=F: process the data using snapshot mode, which will process item supplier location data based on a full snapshot of all data.
- LANGUAGE refers to which language is to be used. The default is set to ‘en’ for English.
  - LANGUAGE=bpt: Brazilian Portuguese
  - LANGUAGE=zhs: Chinese (Simplified)
  - LANGUAGE=zht: Chinese (Traditional)
  - LANGUAGE=en: English
  - LANGUAGE=fr: French
  - LANGUAGE=de: German
  - LANGUAGE=ja: Japanese
  - LANGUAGE=ko: Korean
  - LANGUAGE=es: Spanish

Program Features

RETL programs use one return code to indicate successful completion. If the program successfully runs, a zero (0) is returned. If the program fails, a non-zero is returned.

Program Status Control Files

To prevent a program from running while the same program is already running against the same set of data, the RMSE code utilizes a program status control file. At the beginning of each module, dwi_config.env is run. It checks for the existence of the program status control file. If the file exists, then a message stating, ‘${PROGRAM_NAME} has already started’, is logged and the module exits. If the file does not exist, a program status control file is created and the module executes. If the module fails at any point, the program status control file is not removed, and the user is responsible for removing the control file before re-running the module.

File Naming Conventions

The naming convention of the program status control file allows a program whose input is a text file to be run multiple times at the same time against different files.

The name and directory of the program status control file is set in the configuration file (dwi_config.env). The directory defaults to $MMHOME/error. The naming convention for the program status control file itself defaults to the following dot separated file name:

- The program name
- The first filename, if one is specified on the command line
- ‘status’
- The business virtual date for which the module was run

For example, the program status control file for the invildex program would be named as follows for the VDATE of March 21, 2004:

$MMHOME/error/invildex.invilddm.txt.status.20040321
Restart and Recovery

Because RETL processes all records as a set, as opposed to one record at a time, the method for restart and recovery must be different from the method that is used for Pro*C. The restart and recovery process serves the following two purposes:

1. It prevents the loss of data due to program or database failure.
2. It increases performance when restarting after a program or database failure by limiting the amount of reprocessing that needs to occur.

Most modules use a single RETL flow and do not require the use of restart and recovery. If the extraction process fails for any reason, the problem can be fixed, and the entire process can be run from the beginning without the loss of data. For a module that takes a text file as its input, the following two choices are available that enable the module to be re-run from the beginning:

1. Re-run the module with the entire input file.
2. Re-run the module with only the records that were not processed successfully the first time and concatenate the resulting file with the output file from the first time.

To limit the amount of data that needs to be re-processed, more complicated modules that require the use of multiple RETL flows utilize a bookmark method for restart and recovery. This method allows the module to be restarted at the point of last success and run to completion. The bookmark restart/recovery method incorporates the use of a bookmark flag to indicate which step of the process should be run next. For each step in the process, the bookmark flag is written to and read from a bookmark file.

Note: If the fix for the problem causing the failure requires changing data in the source table or file, then the bookmark file must be removed and the process must be re-run from the beginning in order to extract the changed data.

Bookmark File

The name and directory of the restart and recovery bookmark file is set in the configuration file (dwi_config.env). The directory defaults to $MMHOME/rfx/bookmark. The naming convention for the bookmark file itself defaults to the following 'dot'-separated file name:

- The program name
- The first filename, if one is specified on the command line
- ‘bkm’
- The business virtual date for which the module was run

The example below illustrates the bookmark flag for the invildex program run on the VDATE of January 5, 2004:

$MMHOME/rfx/bookmark/invildex.invilddm.txt.bkm.20040105

Message Logging

Message logs are written daily in a format described in this section.
Daily Log File

Every RETL program writes a message to the daily log file when it starts and when it finishes. The name and directory of the daily log file is set in the configuration file (dwi_config.env). The directory defaults to $MMHOME/log. All log files are encoded UTF-8.

The naming convention of the daily log file defaults to the following ‘dot’ separated file name:

- The business virtual date for which the modules are run
- ‘.log’

For example, the location and the name of the log file for the business virtual date (VDATE) of March 21, 2004 would be the following:

$MMHOME/log/20040321.log

Format

As the following examples illustrate, every message written to a log file has the name of the program, a timestamp, and either an informational or error message:

invildex 16:22:52: Program starting...
invildex 16:22:52: Step1 - process current day data change
invildex 16:22:59: Analyze table rmsint110buser1.GET_ITEM_MASTER_TEMP
invildex 16:22:59: Step2 - Stock-on-hand and in-transit info
invildex 16:23:04: Analyze table rmsint110buser1.GET_ITEM_LOC_TEMP
invildex 16:23:04: Step3 - process on-order quantity and unit cost
invildex 16:23:12: Analyze table rmsint110buser1.FINAL_COMP_ITEM_ON_ORDER_TEMP
invildex 16:23:12: Step4 - Process on-order and original item/loc information
invildex 16:23:18: Drop table rmsint110buser1.GET_ITEM_LOC_TEMP
invildex 16:23:19: Drop table rmsint110buser1.GET_ITEM_MASTER_TEMP
invildex 16:23:19: Drop table rmsint110buser1.FINAL_COMP_ITEM_ON_ORDER_TEMP
invildex 16:23:19: Number of records in /projects/dwi11.0/dev/data/invilddm.txt = 37
invildex 16:23:19: Program completed successfully

If a program finishes unsuccessfully, an error file is usually written that indicates where the problem occurred in the process. There are some error messages written to the log file, such as ‘No output file specified’, that require no further explanation written to the error file.
Program Error File

In addition to the daily log file, each program also writes its own detail flow and error messages. Rather than clutter the daily log file with these messages, each program writes out its errors to a separate error file unique to each execution.

The name and directory of the program error file is set in the configuration file (dwi_config.env). The directory defaults to $MMHOME/error. All errors and all routine processing messages for a given program on a given day go into this error file (for example, it will contain both the stderr and stdout from the call to RETL). All error files are encoded UTF-8.

The naming convention for the program’s error file defaults to the following ‘dot’ separated file name:

- The program name
- The first filename, if one is specified on the command line
- The business virtual date for which the module was run

For example, all errors and detail log information for the invildex program would be placed in the following file for the batch run of March 21, 2004:

$MMHOME/error/invildex.invilddm.txt.20040321

RMSE Reject Files

RMSE extract modules may produce a reject file if they encounter data related problems, such as an inability to find data on required lookup tables. The module tries to process all data and then indicates that records were rejected so that all data problems can be identified in one pass and corrected; then, the module can be re-run to successful completion. If a module does reject records, the reject file is not removed, and the user is responsible for removing the reject file before re-running the module.

The records in the reject file contain an error message and key information from the rejected record. The following example illustrates a record that is rejected due to problems within the currency conversion library:

Unable to convert currency for LOC_IDNT, DAY_DT|3|20011002

The name and directory of the reject file is set in the configuration file (dwi_config.env). The directory defaults to $MMHOME/data.

---

Note: A directory specific to reject files can be created. The dwi_config.env file would need to be changed to point to that directory.
Schema Files

RETL uses schema files to specify the format of incoming or outgoing datasets. The schema file defines each column’s data type and format, which is then used within RETL to format/handle the data. For more information about schema files, see the latest RETL Programmer’s Guide. Schema file names are hard-coded within each module since they do not change on a day-to-day basis. All schema files end with “.schema” and are placed in the “$MMHOME/rfx/schema” directory.

Resource Files

RMSE Kornshell programs use resource files so that the same RETL programs can run in various language environments. For each language, there is one resource file. Resource files contain hard-coded strings that are used by extract programs. The name and directory of the resource file is set in the configuration file (dwi_config.env). The default directory is ${MMHOME}/rfx/include.

The naming convention for the resource file follows the two-letter ISO code standard abbreviation for languages (for example, en for English, fr for French, ja for Japanese, es for Spanish, de for German, and so on).

Command Line Parameters

A module handles command line parameters in one of the three ways described in this section.

**Note:** For some modules, default output file names and schema names correspond to RDW program names.

Modules That Do Not Require Parameters

Some RMSE extraction modules do not require passing in any parameters. The output path/filename defaults to $DATA_DIR/(RDW program name).txt. Similarly, the schema format for the records in these files are specified in the file - $SCHEMA_DIR/(RDW program name).schema.

Non-File Based Modules That Require Parameters

In order for some non-file based RETL modules to run, command line parameters need to be passed in at the UNIX command line. These RMSE modules require an output_file_path and output_file_name to be passed in. These modules may allow the operator to specify more than one output file.

For example:

```
invildex.ksh output_file_path/output_file_name
```

ReSA Ffile-Based Modules That Require Parameters

In order for some file-based RETL modules to run, command line parameters need to be passed in at the UNIX command line. ReSA file-based modules require the following to be passed in:

- output_file_path and output_file_name
- input_file_path and input_file_name

For example:

```
lptotldex output_file_path/output_file_name input_file_path/input_file_name
```
Multi-Threading For RMSE ReSA Modules

In contrast to the way in which multi-threading is defined in UNIX, RMSE modules use ‘multi-threading’ to refer to the running of a single RETL program multiple times on separate groups of data simultaneously. Multi-threading is only available for RMSE ReSA extraction modules that take a text file as input. Depending upon how it is implemented, multi-threading can reduce the total amount of processing time.

File-based extraction modules have to be run once for each input file. A different output file must be specified for each input file. It is the responsibility of the client to set up, as part of the daily batch operation, a process to combine all the resulting text files into one file using the UNIX concatenation (‘cat’) command.

The example below represents a scenario in which the lptotldex.ksh module is run three times for three input files.

```
   lptotldex ${MMHOME}/data/lptotlddm.1000000009
   ${MMHOME}/data/RDWS_1000000009_20020310_20020311
   lptotldex ${MMHOME}/data/lptotlddm.1000000010
   ${MMHOME}/data/RDWS_1000000010_20020310_20020311
   lptotldex ${MMHOME}/data/lptotlddm.1000000011
   ${MMHOME}/data/RDWS_1000000011_20020310_20020311
```

To concatenate the three output files, run the following command in the ${MMHOME}/data directory:

```
cat lptotlddm.1000000009 lptotlddm.1000000010 lptotlddm.1000000011 > lptotlddm.txt
```

In this example, lptotlddm.txt becomes the combined text file.

Typical Run and Debugging Situations

The following examples illustrate typical run and debugging situations for types of programs. The log, error, and so on file names referenced below assume that the module is run on the business virtual date of March 9, 2004. See the previously described naming conventions for the location of each file.

For example:

To run invildex.ksh:

1. Change directories to $MMHOME/rfx/src.
2. At a UNIX prompt enter:
   ```
   % invildex.ksh $MMHOME/data/invilddm.txt
   ```

If the module runs successfully, the following results:

1. **Log file**: Today’s log file, 20040309.log, contains the messages “Program started …” and “Program completed successfully” for invildex.ksh.
2. **Data**: The invilddm.txt file exists in the $MMHOME/data directory and contains the extracted records.
3. **Error file**: The program’s error file, invildex.invilddm.txt.20040309, contains the standard RETL flow (ending with “All threads complete” and “Flow ran successfully”) and no additional error messages.
4. **Program status control**: The program status control file, invildex.invilddm.txt.status.20040309, does not exist.
5. **Reject file**: The reject file, invildex.invilddm.txt.rej.20040309, does not exist.
If the module does not run successfully, the following results:

1. **Log file:** Today’s log file, 20040309.log, does not contain the “Program completed successfully” message for invildex.ksh.
2. **Data:** The invilddm.txt file may exist in the data directory but may not contain all the extracted records.
3. **Error file:** The program’s error file, invildex.invilddm.txt.20040309, may contain an error message.
4. **Program status control:** The program status control file, invildex.invilddm.txt.status.20040309, exists.
5. **Reject file:** The reject file, invildex.invilddm.txt.rej.20040309, does not exist because this module does not reject records.
6. **Bookmark file (in certain conditions):** The bookmark file, invildex.invilddm.txt.bkm.20040309, exists because this module contains more than one flow. The error occurred after the first flow (for example, during the second flow).

To re-run a module from the beginning, perform the following actions:

1. Determine and fix the problem causing the error.
2. Remove the program’s status control file.
3. Remove the bookmark file from $MMHOME/rfx/bookmark
4. Change directories to $MMHOME/rfx/src. At a UNIX prompt, enter:
   ```
   %invildex.ksh $MMHOME/data/invilddm.txt
   ```

**Note:** To understand how to engage in the restart and recovery process, see the section, ‘Restart and recovery’ earlier in this chapter.

---

**Running the Time 454 Extract Module**

The time 454 extract module requires the steps below to run successfully:

1. Log in to the RMS database server as RMS RETL-specific database user. Run the profile and verify that the MMUSER and PASSWORD variables are set to the RETL-specific database user, and the appropriate password. Verify the RETL executable is in the path of your UNIX session by typing:
   ```
   %which rfx
   ```
2. Change directories to $MMHOME/install.
3. Modify the variable l_path in the extract_time.sql script to reference the UTL_FILE directory specified in the RMS database parameter file.
4. At the UNIX prompt enter:
   ```
   %extract_time.ksh
   ```
   This script generates three files called time_454*.txt, wkday*.txt, and start_of_half_month*.txt located in the utl_file_dir directory specified in your RMS database parameter file.
5. Change directories on the UNIX server to $MMHOME/log. Review the log file that was created or modified.
6. Change directories on the UNIX server to $MMHOME/error. Review the error files that were created.
7. Move the three output files to $MMHOME/install directory.
RETL Program Overview for the RMS-RPAS Interface

Oracle Retail ETL Architecture

RMS works in conjunction with the Oracle Retail Extract Transform and Load (RETL) framework. This architecture utilizes a high performance data processing tool that allows database batch processes to take advantage of parallel processing capabilities.

The RETL framework runs and parses through the valid operators composed in XML scripts.

More information about the RETL tool is available in the latest RETL Programmer’s Guide.

The diagram below illustrates the extraction processing architecture. Instead of managing the change captures as they occur in the source system during the day, the process involves extracting the current data from the source system. The extracted data is output to flat files. These flat files are then available for consumption by products such as RPAS.

The target system has its own way of completing the transformations and loading the necessary data into its system, where it can be used for further processing in the environment. See RPAS documentation for information related to transformations and loadings.

The architecture relies upon two distinct stages, shown in the diagram below. Stage 1 is the extraction from the RMS database using well-defined flows specific to the RMS database. The resulting output is comprised of data files written in a well-defined schema file format. This stage includes no destination specific code.

Stage 2 introduces a flow specific to the destination. In this case, flows for RPAS are designed to transform the data so that RPAS can import the data properly.
The two stages of RETL processing

**RETL Program Overview**

This section summarizes the RETL program features utilized in the RMS extractions and loads. Installation information about the RETL tool is available in the latest RETL Programmer’s Guide.

**Configuration**

**Version of RETL**

Before trying to configure and run RMS RETL, install RETL version 12.0 or later, which is required to run RMS RETL. See the latest RETL Programmer’s Guide for thorough installation information.

**RETL User and Permissions**

The permissions are set up as per the RETL Programmer’s Guide. RMS RETL reads and writes data files and creates, deletes, updates and inserts into tables. If these permissions are not set up properly, extractions fail.
**Environment Variables**

See the RETL Programmer’s Guide for RETL environment variables that must be set up for your version of RETL. You will need to set RDF_HOME to your base directory for RMS RETL. This is the top level directory that you selected during the installation process. In .profile, you should add a line such as the following:

```
export RDF_HOME=<base directory for RMS RETL>
```

**rmse_rpas_config.env Settings for RPAS**

There are several constants that must be set in rmse_rpas_config.env depending upon a retailer’s preferences and the local environment. These are summarized in the following table.

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Default Value</th>
<th>Alternate Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE_TYPE</td>
<td>vdate</td>
<td>current_date</td>
<td>Determines whether the date used in naming the error, log, and status files is the current date or the VDATE value found in the PERIOD table.</td>
</tr>
<tr>
<td>DBNAME</td>
<td>rtkdev01</td>
<td>Depends on installation</td>
<td>The database schema name.</td>
</tr>
<tr>
<td>RMS_OWNER</td>
<td>RPASINT</td>
<td>Depends on installation</td>
<td>The username of the RMS database schema owner.</td>
</tr>
<tr>
<td>BA_OWNER</td>
<td>Depends on installation</td>
<td>The username of the RMS batch user (not currently used by RMS-RPAS).</td>
<td></td>
</tr>
<tr>
<td>CONN_TYPE</td>
<td>thin</td>
<td>oci</td>
<td>The way in which RMS connects to the database.</td>
</tr>
<tr>
<td>DBHOST</td>
<td>mspdev17</td>
<td>Depends on installation</td>
<td>The computer hardware node name.</td>
</tr>
<tr>
<td>DBPORT</td>
<td>1524</td>
<td>Depends on installation</td>
<td>The port on which the database listener resides.</td>
</tr>
<tr>
<td>LOC_ATTRIBUTES_ACTIVE</td>
<td>False</td>
<td>True</td>
<td>Determines whether rmse_rpas_attributes.ksh is run or not.</td>
</tr>
<tr>
<td>PROD_ATTRIBUTES_ACTIVE</td>
<td>False</td>
<td>True</td>
<td>Determines whether rmse_rpas_attributes.ksh is run or not.</td>
</tr>
<tr>
<td>DIFFS_ACTIVE</td>
<td>True</td>
<td>False</td>
<td>Determines whether rmse_rpas_merchhier.ksh generates data files that contain diff allocation information.</td>
</tr>
</tbody>
</table>
### Constant Name

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Default Value</th>
<th>Alternate Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISSUES_ACTIVE</td>
<td>True</td>
<td>False</td>
<td>If set to ‘True’, rmse_rpas_stock_on_ha nd also extracts stock at the warehouse level. If set to ‘False’, rmse_rpas_stock_on_ha nd extracts stock at the store level only.</td>
</tr>
<tr>
<td>LOAD_TYPE</td>
<td>CONVENTIONAL</td>
<td>DIRECT</td>
<td>Data loading method to be used by SQL*Loader (Direct may be faster than conventional.)</td>
</tr>
<tr>
<td>DB_ENV</td>
<td>ORA</td>
<td>DB2, TERA</td>
<td>Database type (Additional changes to the software may be needed if a database other than Oracle is selected.)</td>
</tr>
<tr>
<td>NO_OF_CPUS</td>
<td>4</td>
<td>Depends on installation</td>
<td>Used in parallel database query hints to improve performance.</td>
</tr>
<tr>
<td>LANGUAGE</td>
<td>en</td>
<td>Various</td>
<td>En = English</td>
</tr>
<tr>
<td>RFX_OPTIONS</td>
<td>-c $RDF_HOME/rfx/etc/rfx.conf -s SCHEMAFILE</td>
<td>-c $RDF_HOME/rfx/etc/rfx.conf -s SCHEMAFILE</td>
<td>Processing speed may be increased for some extractions if the -s SCHEMAFILE option is omitted</td>
</tr>
</tbody>
</table>

You must also set up the environment variable PASSWORD in the rmse_rpas_config.env, .kshrc or some other location that can be referenced. In the example below, adding the line to the rmse_rpas_config.env causes the password ‘mypasswd’ to be used to log into the database:

```bash
export PASSWORD=mypasswd
```

Be sure to review the environmental parameters in the rmse_rpas_config.env file before executing batch modules.

### Steps to Configure RETL

1. Log in to the UNIX server with a UNIX account that will run the RETL scripts.
2. Change directories to `<base_directory>/rfx/etc`.
3. Modify the constants from the table above in the rmse_rpas_config.env script as needed.
Program Return Code

RETL programs use a return code to indicate successful completion. If the program successfully runs, a zero (0) is returned. If the program fails, a non-zero is returned.

Program Status Control Files

To prevent a program from running while the same program is already running against the same set of data, the code utilizes a program status control file. At the beginning of each module, rmse_rpas_config.env is run. This script checks for the existence of the program status control file. If the file exists, then a message stating, ‘${PROGRAM_NAME} has already started’, is logged and the module exits. If the file does not exist, a program status control file is created and the module executes.

If the module fails at any point, the program status control file is not removed, and the user is responsible for removing the control file before re-running the module.

File Naming Conventions

The name and directory of the program status control file is set in the configuration script (rmse_rpas_config.env). The directory defaults to $RDF_HOME/error. The naming convention for the program status control file itself defaults to the following dot separated file name:

- The program name
- ‘status’
- The business virtual date for which the module was run

For example, a program status control file for the rmse_rpas_daily_sales.ksh program would be named as follows for a batch run on the business virtual date of January 5, 2001:

$RDF_HOME/error/rmse_rpas_daily_sales.status.20010105

Restart and Recovery

Because RETL processes all records as a set, as opposed to one record at a time, the method for restart and recovery must be different from the method that is used for Pro*C. The restart and recovery process serves the following two purposes:

1. It prevents the loss of data due to program or database failure.
2. It increases performance when restarting after a program or database failure by limiting the amount of reprocessing that needs to occur.

The RMS extract (RMSE) modules extract from a source transaction database or text file and write to a text file. The RMS load module imports data from flat files, performs transformations if necessary, and then loads the data into the applicable RMS table.

Most modules use a single RETL flow and do not require the use of restart and recovery. If the extraction process fails for any reason, the problem can be fixed, and the entire process can be run from the beginning without the loss of data. No RMS to RPAS extraction programs have any restart/recovery capability. The single RMS load program, rmsl_rpas_forecast.ksh, takes a text file as its input, and the following two choices are available that enable the program to complete the load in the event of an error:

- Re-run the program with the entire input file.
- Re-run the program with only the input records that were not processed successfully the first time.
Message Logging

Message logs are written daily in a format described in this section.

Daily Log File

Every RETL program writes a message to the daily log file when it starts and when it finishes. In some cases, progress messages are also written. The name and directory of the daily log file is set in the configuration script (rmse_rpas_config.env). The directory defaults to $RDF_HOME/log. All log files are encoded UTF-8.

The naming convention of the daily log file defaults to the following “dot” separated file name:

- The business virtual date for which the modules are run
- ‘.log’

For example, the location and the name of the log file for the business virtual date of January 5, 2001 would be the following:

$RDF_HOME/log/20010105.log

Format

As the following examples illustrate, every message written to a log file has the name of the program, a timestamp, and either an informational or error message. For example:

rmse_rpas_item_retail 17:09:07: Program started ...
rmse_rpas_item_retail 17:09:12: Program completed successfully

Some error messages are also written to the log file, such as ‘No output file specified’.

Program Error File

In addition to the daily log file, each program also writes its own detailed flow and error messages. Rather than clutter the daily log file with these messages, each program writes out its errors to a separate error file unique to each execution.

If a program finishes unsuccessfully, a message is usually written in the error file that indicates where the problem occurred in the process.

The name and directory of the program error file is set in the applicable configuration file (rmse_rpas_config.env). The directory defaults to $RDF_HOME/error. All errors and all routine processing messages for a given program on a given day go into this error file (for example, it will contain both the stderr and stdout produced during execution of the program).

The naming convention for the program’s error file defaults to the following “dot” separated file name:

- The program name
- The business virtual date for which the module was run

For example, all errors and detailed log information for the rms_item_master.ksh program would be placed in the following file for the batch run on the business virtual date of January 5, 2001:

$MMHOME/error/rms_item_master.20010105
RMSE Reject Files

RMSE extract modules may produce a reject file if they encounter data related problems, such as the inability to find data on required lookup tables. The module tries to process all data and then indicates that records were rejected so that all data problems can be identified in one pass and corrected; then, the module can be re-run to successful completion. If a module does reject records, the reject file is not removed, and the user is responsible for removing the reject file before re-running the module. The records in the reject file consist of the rejected records.

The name and directory of the reject file are defined in the applicable configuration script (rmse_rpas_config.env). The directory defaults to $RDF_HOME/data.

---

**Note:** A directory specific to reject files can be created. The \texttt{rmse_rpas_config.env} script would need to be changed to define the reject directory constant such that it would point to that directory.

---

The naming convention for the reject file defaults to the following “dot” separated file name:

- The program name
- The first filename, if one is specified on the command line
- ‘rej’
- The business virtual date for which the module was run

Schema Files Overview

RETL uses schema files to specify the format of incoming or outgoing datasets. The schema file defines each column’s data type and format, which is then used within RETL to format/handle the data. For more information about schema files, see the latest RETL Programmer’s Guide. Schema file names are hard-coded within each module because they do not change on a day-to-day basis. All schema files end with “.schema” and are placed in the “$RDF_HOME/rfx/schema” directory.
Command Line Parameters

The only programs or scripts that allow command line parameters (or arguments) are the rmse_rpas_config.env script and the pre_rmse_rpas.ksh and rmse_rpas.ksh programs. All of the command line parameters for these modules are optional and are described below (the square brackets indicate that the parameter is optional):

**rmse_rpas_config.env**

Usage: $RDF_HOME/rfx/etc/rmse_rpas_config.env [-t $*] [-r $*] [-s $*] [-v $* | -c $*]

Description of Command Line Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-t:</td>
<td>This option causes rmse_rpas_config.env to skip the initializing of the environment variables that obtain their values from the <code>.txt</code> files, except for VDATE which is initialized with the date found in the vdate.txt file. This option is utilized by pre_rmse_rpas.ksh, rmse_rpas.ksh, rdft.ksh and outage.ksh when they call rmse_rpas_config.env.</td>
</tr>
<tr>
<td>-r:</td>
<td>This option prevents the redirection of all output (stdout and stderr) to the error file. This can be useful during debugging and maintenance. This option can also be utilized by rmse_rpas.ksh, rdft.ksh and outage.ksh when they call rmse_rpas_config.env.</td>
</tr>
<tr>
<td></td>
<td>The ‘-t’ and ‘-r’ options must be followed by ‘$<em>’ on the line which invokes this script. This step is necessary in order to preserve the command line arguments or options that may have been present on the command line for the RETL script that invokes this script. However, the ‘$</em>’ should only appear once if both options are used.</td>
</tr>
<tr>
<td>-s:</td>
<td>This option causes rmse_rpas_config.env to skip the STATUS_FILE test. This is also useful during maintenance and debugging.</td>
</tr>
<tr>
<td>-v:</td>
<td>If DATE_TYPE (in rmse_rpas_config.env) is set to ‘vdate’, this option prevents the normal exit with an error message when the vdate.txt file is empty or non-existent; instead, it will use the current date to derive FILE_DATE. However, if DATE_TYPE is set to ‘vdate’, and vdate.txt actually does exist and is non-empty, the date in vdate.txt continues to be used even if this option is set. If DATE_TYPE is set to ‘current_date’, this option has no effect.</td>
</tr>
<tr>
<td>-c:</td>
<td>This option overrides the DATE_TYPE switch setting and causes the current date to be used to derive FILE_DATE regardless of what DATE_TYPE is set to. This option is utilized by pre_rmse_rpas.ksh when it calls rmse_rpas_config.env, if it is run with the -c option on its command line. The ‘-c’ option is normally only used when rmse_rpas_config.env is called from pre_rmse_rpas.ksh.</td>
</tr>
</tbody>
</table>
|        | If only one command line option is used, it must be followed by ‘$*’. But if more than one option is specified, then ‘$*’ must be entered on the command line only once after all options have been entered. The ‘$*’ is necessary in order to preserve the command line arguments or options (if there are any) that are present on the command line that is used to execute the RETL script which invokes this script.
If more than one option is specified, options must appear on the command line in the same order as shown on the “Usage” line, above.

**pre_rmse_rpas.ksh**

- **Usage:** `pre_rmse_rpas.ksh [-c]

The ‘-c’ option is used to specify what option is to be placed on the command line when it is called by this program. It is usually used the first time that `pre_rmse_rpas.ksh` is run at a new installation or if the state of the `vdate.txt` file is unknown. This option is passed directly to `rmse_rpas_config.env` when it is called by `pre_rmse_rpas.ksh`. No other use is made of this parameter by `pre_rmse_rpas.ksh`.

This option causes `rmse_rpas_config.env` to use the current date to initialize `FILE_DATE` instead of possibly setting it to `VDATE`, which is obtained from the `vdate.txt` file. (`FILE_DATE` is the date that is used to name the error, log, and status files.)

The current date is used regardless of how `DATE_TYPE` is set in `rmse_rpas_config.env`. By using the ‘-c’ option, there is no need to manually set up the `vdate.txt` file before running this script.

The normal mode for `pre_rmse_rpas.ksh` (without the -c option) is that when it calls `rmse_rpas_config.env`, `FILE_DATE` is set to `VDATE` or the current date, depending on how `DATE_TYPE` is set in `rmse_rpas_config.env`. If `DATE_TYPE` is set to ‘vdate’, and if the `vdate.txt` file does not exist or is empty, `rmse_rpas_config.env` (and this program) exits with an error message.

The use of this option does not affect what date is used by any of the other RETL scripts that run after this script is done. After `pre_rmse_rpas.ksh` has run, when the other RETL scripts are run, they call `rmse_rpas_config.env` with no options on the command line, and their files are named using `VDATE` or the current date, depending on how `DATE_TYPE` is set in `rmse_rpas_config.env`.

**rmse_rpas.ksh**

- **Usage:** `rmse_rpas.ksh [-c]

The presence of the ‘-c’ option causes `FILE_DATE` in `rmse_rpas_config.env` to be set to the current date instead of possibly using `VDATE` (which gets its value from the `vdate.txt` file), but only when it is called by `rmse_rpas.ksh` and `pre_rmse_rpas.ksh` (pre_rmse_rpas.ksh is invoked by rmse_rpas.ksh). It has no effect when other extract programs call `rmse_rpas_config.env`, at the time that they are invoked by `rmse_rpas.ksh`. This option is passed directly to `rmse_rpas_config.env` and `pre_rmse_rpas.ksh` when they are called by `rmse_rpas.ksh`. No other use is made of this parameter by `rmse_rpas.ksh`.

**RMSE I/O File Names**

Most of the output path/filenames have the format, `$DATA_DIR/(RMSE_RPAS_program name).dat`. Similarly, the schema format for the records in these files are specified in the file - `$SCHEMA_DIR/(RMSE_RPAS_program name).schema`. 
Typical Run and Debugging Situations

The following examples illustrate typical run and debugging situations for programs. The log, error, etc. file names referenced below assume that the module is run on the business virtual date of March 9, 2001. See the previously described naming conventions for the location of each file.

For example:

To run rmse_rpas_stores.ksh:
1. Change directories to $RDF_HOME/rfx/src.
2. At a UNIX prompt ($) enter:
   $ rmse_rpas_stores.ksh

If the module runs successfully, the following results:

1. **Log file:** Today’s log file, 20010309.log, contains the messages “Program started …” and “Program completed successfully” for rmse_rpas_stores.
2. **Data:** The rmse_rpas_stores.dat file exists in the data directory and contains the extracted records.
3. **Schema:** The rmse_rpas_stores.schema file exists in the schema directory and contains the definition of the data file in #2 above.
4. **Error file:** The program’s error file, rmse_rpas_stores.20010309, contains the standard RETL flow (ending with “All threads complete” and “Flow ran successfully”) and no error messages.
5. **Program status control:** The program status control file, rmse_rpas_stores.status.20010309, will not exist.
6. **Reject file:** The reject file, rmse_rpas_stores.rej.20010309, will not exist.

If the module does not run successfully, the following results:

1. **Log file:** Today’s log file, 20010309.log, does not contain the “Program completed successfully” message for rmse_rpas_stores.
2. **Data:** The rmse_rpas_stores.dat file may exist in the data directory but may not contain all the extracted records.
3. **Schema:** The rmse_rpas_stores.schema file exists in the schema directory and contains the definition of the data file in #2 above.
4. **Error file:** The program’s error file, rmse_rpas_stores.20010309, may contain one or more error messages.
5. **Program status control:** The program status control file, rmse_rpas_stores.status.20010309, exists.
6. **Reject file:** The reject file, rmse_rpas_stores.status.20010309, does not exist because this module does not reject records.

To re-run the module, perform the following actions:

1. Determine and fix the problem causing the error.
2. Remove the program’s status control file.
3. Change directories to $RDF_HOME/rfx/src. At a UNIX prompt, enter:
   $ rmse_rpas_stores.ksh
Internationalization is the process of creating software that is able to be translated more easily. Changes to the code are not specific to any particular market. RMS has been internationalized to support multiple languages.

This section describes configuration settings and features of the software that ensure that the base application can handle multiple languages.

Translation
Translation is the process of interpreting and adapting text from one language into another. Although the code itself is not translated, components of the application that are translated may include the following, among others:

- Graphical user interface (GUI)
- Error messages

The user interface for RMS 13.0 has been translated into:

- German
- French
- Spanish
- Japanese
- Traditional Chinese
- Simplified Chinese
- Korean
- Brazilian Portuguese
- Russian
- Italian

RMS Client Language Settings
The client sets up the user’s language preferences. RMS determines the user’s language setting and displays the code string associated with it. RMS has a fail/safe mechanism built into the code. If the user’s preference language string is not found, then RMS rolls back to English.

**Note:** A retailer has the two options below regarding internationalization when installing the application. See the RMS Installation Guide for the procedures related to each.

- English and multiple secondary languages
- Install English first and then update with a translated language (fully translated non-English installation)
Key RMS Tables Related to Internationalization

Several new tables were created to handle displayable text that can also be translated. If the retailer creates a new form, a new menu, or a new object on a form, then the retailer will need to populate these tables with the corresponding information. If the retailer customizes the information in any of the tables FORM_ELEMENTS, FORM_ELEMENTS_LANGS, MENU_ELEMENTS, or MENU_ELEMENTS_LANGS, the base_ind field in customized records must contain ‘N’. Any record with BASE_IND=N will be preserved in a temp table during future patches.

**FORM_ELEMENTS**

This table is used for screen display and holds the master list of items for all forms whose labels/prompts are translated. This information will always be in English. The BASE_IND=Y means that the item is part of the base Oracle Retail code set. BASE_IND=N indicates that the item was added as part of retailer customization. Anything with the BASE_IND=N will be preserved at upgrade time on the FORM_ELEMENTS_TEMP, but the retailer is responsible for moving the data back to FORM_ELEMENTS.

**FORM_ELEMENTS_LANGS**

This table is used for screen display. This table holds translated values for labels/prompts on forms. This information will be in a language that is defined on the lang column of the user_attrib table. All users see data from this table, as the retailer may customize the text of a given field. The access key for a button is defined by filling in the DEFAULT_ACCESS_KEY field. At runtime, that character will be marked in the string, and function as the access key. Any time the retailer changes the DEFAULT_LABEL_PROMPT or DEFAULT_ACCESS_KEY, the BASE_IND should be updated to N because it is not part of the base language translations provided by Oracle Retail. Anything with the BASE_IND=N will be preserved at upgrade time on the FORM_ELEMENTS_LANGS_TEMP, but the retailer is responsible for moving the data back to FORM_ELEMENTS_LANGS.

**MENU_ELEMENTS**

This table is used for screen display. This table holds the master list for all menus whose items are translated. This information will always be in English. The access key for a menu option is defined by using the ampersand (&) before the character that is the access key in the default description. The BASE_IND=Y means that the item is part of the base Oracle Retail code set. BASE_IND=N indicates that the item was added as part of retailer customization. Anything with the BASE_IND=N will be preserved at upgrade time on the MENU_ELEMENTS_TEMP, but the retailer is responsible for moving the data back to MENU_ELEMENTS.

**MENU_ELEMENTS_LANGS**

This table is used for screen display. This table holds the values for all menus whose items are translated. This information will be in a language that is defined on the lang table. Even English language users see data from this table, as the retailer may customize the text of a given menu option. Any time the retailer changes the LANG_LABEL, the BASE_IND should be updated to N because it is not part of the base language translations provided by Oracle Retail. Anything with the BASE_IND=N will be
preserved at upgrade time on the MENU_ELEMENTS_LANGS_TEMP, but the retailer is responsible for moving the data back to MENU_ELEMENTS_LANGS.

FORM_MENU_LINK
This table is used for screen display. This table holds the intersection of form and menu files, mapping each form to the menu that it displays.

CODE_DETAIL_TRANS
This table holds non-primary language descriptions of code types defined on the CODE_DETAIL table. The retailer has a multi-language option.
Custom Post Processing

RMS has an optional method of handling unwanded cartons for customer post processing. This only applies to stock order receiving. An unwanded carton occurs when a carton was not scanned when the stock order was shipped, but is scanned at the time of the receipt. These cartons do not contain any shipment records in RMS.

Since the carton contains items that did not go through the appropriate transfer out procedure, the inventory for those items will not be accurate. As a result, the message which contains the unwanded (unscanned) carton is rejected by RMS to the RIB error hospital at the time of receiving. RMS will then publish to the warehouse management system via the RIB of the unwanded cartons in the RcptAdjustDesc message. The warehouse management system will then send RMS a shipment message containing the appropriate BOL and the carton ID. RMS will process the message and create or update the shipment records. The next time RMS tries to process the rejected receipt message with the unwanded carton, RMS will be able to process it.

The client’s warehouse management system must be able to support the processing of the RcptAdjustDesc message above in order for this functionality of unwanded carton to work successfully.
Configuring RMS for Single Sign-on

Overview

What is Single Sign-On?

Single Sign-On (SSO) is a term for the ability to sign onto multiple web applications via a single user ID/Password. There are many implementations of SSO – Oracle currently provides three different implementations: Oracle Single Sign-On (SSO), Java SSO (with the 10.1.3.1 release of OC4J) and Oracle Access Manager (provides more comprehensive user access capabilities).

Most, if not all, SSO technologies use a session cookie to hold encrypted data passed to each application. The SSO infrastructure has the responsibility to validate these cookies and, possibly, update this information. The user is directed to log on only if the cookie is not present or has become invalid. These session cookies are restricted to a single browser session and are never written to a file.

Another facet of SSO is how these technologies redirect a user’s Web browser to various servlets. The SSO implementation determines when and where these redirects occur and what the final screen shown to the user is.

Most SSO implementations are performed in an application’s infrastructure and not in the application logic itself. Applications that leverage infrastructure managed authentication (such as deploying specifying “Basic” or “Form” authentication) typically have little or no code changes when adapted to work in an SSO environment.

What Do I Need for Oracle Single Sign-On?

The nexus of an Oracle Single Sign-On system is the Oracle Identity Management Infrastructure installation. This consists of the following components:

- An Oracle Internet Directory (OID) LDAP server, used to store user, role, security, and other information. OID uses an Oracle database as the back-end storage of this information.
- An Oracle Single Sign-On servlet, used to authenticate the user and create the SSO session cookie. This servlet is deployed within the infrastructure Oracle Application Server (OAS).
- The Delegated Administration Services (DAS) application, used to administer users and group information. This information may also be loaded or modified via standard LDAP Data Interchange Format (LDIF) scripts.
- Additional administrative scripts for configuring the OSSO system and registering HTTP servers.

Additional OAS servers will be needed to deploy the business applications leveraging the OSSO technology.

Can Oracle Single Sign-On Work with Other SSO Implementations?

Yes, OSSO has the ability to interoperate with many other SSO implementations, but some restrictions exist.
Oracle Single Sign-on Terms and Definitions

Authentication
Authentication is the process of establishing a user’s identity. There are many types of authentication. The most common authentication process involves a user ID and password.

Dynamically Protected URLs
A “Dynamically Protected URL” is a URL whose implementing application is aware of the OSSO environment. The application may allow a user limited access when the user has not been authenticated. Applications that implement dynamic OSSO protection typically display a “Login” link to provide user authentication and gain greater access to the application’s resources.

Identity Management Infrastructure
The Identity Management Infrastructure is the collection of product and services which provide Oracle Single Sign-on functionality. This includes the Oracle Internet Directory, an Oracle HTTP server, and the Oracle Single Sign-On services. The Oracle Application Server deployed with these components is typically referred as the “Infrastructure” instance.

MOD_OSSO
mod_osso is an Apache Web Server module an Oracle HTTP Server uses to function as a partner application within an Oracle Single Sign-On environment. The Oracle HTTP Server is based on the Apache HTTP Server.

Oracle Internet Directory
Oracle Internet Directory (OID) is an LDAP-compliant directory service. It contains user ids, passwords, group membership, privileges, and other attributes for users who are authenticated using Oracle Single Sign-On.

Partner Application
A partner application is an application that delegates authentication to the Oracle Identity Management Infrastructure. One such partner application is the Oracle HTTP Server (OHS) supplied with the Oracle Application Server. OHS uses the MOD_OSSO module to configure this functionality.
All partner applications must be registered with the Oracle Single Sign-On server. An output product of this registration is a configuration file the partner application uses to verify a user has been previously authenticated.

Realm
A Realm is a collection users and groups (roles) managed by a single password policy. This policy controls what may be used for authentication (for example, passwords, X.509 certificates, and biometric devices). A Realm also contains an authorization policy used for controlling access to applications or resources used by one or more applications.
A single OID can contain multiple Realms. This feature can consolidate security for retailers with multiple banners or to consolidate security for multiple development and test environments.
**Statically Protected URLs**

A URL is considered to be “Statically Protected” when an Oracle HTTP server is configured to limit access to this URL to only OSSO authenticated users. Any attempt to access a “Statically Protected URL” results in the display of a login page or an error page to the user.

Servlets, static HTML pages, and JSP pages may be statically protected.

**What Single Sign-On is not**

Single Sign-On is NOT a user ID/password mapping technology. However, some applications can store and retrieve user IDs and passwords for non-SSO applications within an OID LDAP server. An example of this is the Oracle Forms Web Application framework, which maps OSSO user IDs to a database logins on a per-application basis.

**How Oracle Single Sign-On Works**

Oracle Single Sign-On involves a couple of different components. These are:
- The Oracle Single Sign-On (OSSO) servlet, which is responsible for the back-end authentication of the user.
- The Oracle Internet Directory LDAP server, which stores user IDs, passwords, and group (role) membership.
- The Oracle HTTP Server associated with the web application, which verifies and controls browser redirection to the OSSO servlet.
- If the web application implements dynamic protection, then the web application itself is involved with the OSSO system.

**Statically Protected URLs**

When an unauthenticated user accesses a statically protected URL, the following occurs:

1. The Oracle HTTP server recognizes the user has not been authenticated and redirects the browser to the Oracle Single Sign-On servlet.
2. The OSSO servlet determines the user must authenticate, and displays the OSSO login page.
3. The user must sign in via a valid user ID and password. If the OSSO servlet has been configured to support multiple Realms, a valid realm must also be entered. The user ID, password, and realm information is validated against the Oracle Internet Directory LDAP server.
4. The OSSO servlet creates and sends the user’s browser an OSSO session cookie. This cookie is never persisted to disk and is specific only to the current browser session. This cookie contains the user’s authenticated identity. It does NOT contain the user’s password.
5. The OSSO servlet redirects the user back to the Oracle HTTP Server, along with OSSO specific information.
6. The Oracle HTTP Server decodes the OSSO information, stores it with the user’s session, and allows the user access to the original URL.
Dynamically Protected URLs
When an unauthenticated user accesses a dynamically protected URL, the following occurs:

1. The Oracle HTTP server recognizes the user has not been authenticated, but allows the user to access the URL.
2. The application determines the user must be authenticated and sends the Oracle HTTP server a specific status to begin the authentication process.
3. The Oracle HTTP Server redirects the user’s browser session to the OSSO Servlet.
4. The OSSO servlet determines the user must authenticate, and displays the OSSO login page.
5. The user must sign in via a valid user ID and password. If the OSSO servlet has been configured to support multiple Realms, a valid realm must also be entered. The user ID, password, and realm information is validated against the Oracle Internet Directory LDAP server.
6. The OSSO servlet creates and sends the user’s browser an OSSO session cookie. This cookie is never persisted to disk and is specific only to the current browser session. This cookie contains the user’s authenticated identity. It does NOT contain the user’s password.
7. The OSSO servlet redirects the user back to the Oracle HTTP Server, along with OSSO specific information.
8. The Oracle HTTP Server decodes the OSSO information, stores it with the user’s session, and allows the user access to the original URL.

Single Sign-on Topology
Installation Overview

Installing Oracle Single Sign-On consists of installing the following components:

1. Installing the Oracle Internet Directory (OID) LDAP server and the Infrastructure Oracle Application Server (OAS). These are typically performed using a single session of the Oracle Universal Installer and are performed at the same time. OID requires an Oracle relational database and if one is not available, the installer will also install this as well.

   The Infrastructure OAS includes the Delegated Administration Services (DAS) application as well as the OSSO servlet. The DAS application can be used for user and realm management within OID.

2. Installing additional OAS 10.1.2 midtier instances for the Oracle Retail applications, such as RMS, that are based on Oracle Forms technologies. These instances must be registered with the Infrastructure OAS installed in step 1.

3. Installing additional application servers to deploy other Oracle Retail applications and performing application specific initialization and deployment activities.

Infrastructure Installation and Configuration

The Infrastructure installation for OSSO is dependent on the environment and requirements for its use. Deploying an Infrastructure OAS to be used in a test environment does not have the same availability requirements as for a production environment. Similarly, the Oracle Internet Directory (OID) LDAP server can be deployed in a variety of different configurations. See the Oracle Application Server Installation Guide and the Oracle Internet Directory Installation Guide for more details.

OID User Data

Oracle Internet Directory is an LDAP v3 compliant directory server. It provides standards-based user definitions out of the box.

The current version of Oracle Single Sign-On only supports OID as its user storage facility. Customers with existing corporate LDAP implementations may need to synchronize user information between their existing LDAP directory servers and OID. OID supports standard LDIF file formats and provides a JNDI compliant set of Java classes as well. Moreover, OID provides additional synchronization and replication facilities to integrate with other corporate LDAP implementations.

Each user ID stored in OID has a specific record containing user specific information. For role-based access, groups of users can be defined and managed within OID. Applications can thus grant access based on group (role) membership saving administration time and providing a more secure implementation.

OID with Multiple Realms

OID and OSSO can be configured to support multiple user Realms. Each realm is independent from each other and contains its own set of user IDs. As such, creating a new realm is an alternative to installing multiple OID and Infrastructure instances.

Hence, a single Infrastructure OAS can be used to support many development and test environments by defining one realm for each environment.

Realms may also be used to support multiple groups of external users, such as those from partner companies. For more information on Realms, see the Oracle Internet Directory Administrators Guide.
User Management

User Management consists of displaying, creating, updating or removing user information. There are two basic methods of performing user management: LDIF scripts and the Delegate Administration Services (DAS) application.

OID DAS

The DAS application is a web based application designed for both administrators and users. A user may update their password, change their telephone number of record, or modify other user information. Users may search for other users based on partial strings of the user’s name or ID. An administrator may create new users, unlock passwords, or delete users.

The DAS application is fully customizable. Administrators may define what user attributes are required, optional or even prompted for when a new user is created. Furthermore, the DAS application is secure. Administrators may also what user attributes are displayed to other users. Administration is based on permission grants, so different users may have different capabilities for user management based on their roles within their organization.

LDIF Scripts

Script based user management can be used to synchronize data between multiple LDAP servers. The standard format for these scripts is the LDAP Data Interchange Format (LDIF). OID supports LDIF script for importing and exporting user information. LDIF scripts may also be used for bulk user load operations.

User Data Synchronization

The user store for Oracle Single Sign-On resides within the Oracle Internet Directory (OID) LDAP server. Oracle Retail applications may require additional information attached to a user name for application-specific purposes and may be stored in an application-specific database. Currently, there are no Oracle Retail tools for synchronizing changes in OID stored information with application-specific user stores. Implementers should plan appropriate time and resources for this process. Oracle Retail strongly suggests that you configure any Oracle Retail application using an LDAP for its user store to point to the same OID server used with Oracle Single Sign-On.

Setting up RMS for Single Sign-on

To set up Forms for Single Sign-on, the Forms framework must know and/or be configured to use SSO. To do this, the Forms framework configuration file formsweb.cfg must be configured to enable SSO and the mid-tier HTTP Server must be registered with the Oracle Single Sign-On server. In addition, the Forms framework uses Resource Access Descriptor (RAD), to map OSSO user IDs to Database connect strings.
Configuring `formsweb.cfg`

For each Forms application instance there are two attributes in the Forms framework configuration file `formsweb.cfg` that control SSO behavior:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ssoMode</code></td>
<td>true/false</td>
<td>Enables/disables SSO</td>
</tr>
<tr>
<td><code>ssoDynamicResourceCreate</code></td>
<td>true/false</td>
<td>Enables/disables the dynamic RAD entry creation</td>
</tr>
</tbody>
</table>

Creating a RAD Entry

There are three ways by which a RAD entry (mapping an OSSO user ID to a Database connect string) may be created:

- **Administrator Created**
  
  An administrator uses the Delegated Administration Services (DAS) web application that comes with the infrastructure server and that can be launched using the URL `http://<host>:port/oiddas`.

- **User Created**
  
  The user can dynamically create a RAD entry when the Forms framework prompts the user for information. This however requires that the `ssoDynamicResourceCreate` attribute be set to true. If a RAD already exists, the user may also create additional RADs via the DAS application.

- **LDIF Script**
  
  More information about how to use an LDIF script to create a RAD entry may be found by accessing Metalink document 244526.1.
Integrating RMS with Oracle Retail Workspace

The Oracle Retail Workspace installer prompts you to enter the URL for your supported Oracle Retail applications. However, if a client installs a new application after Oracle Retail Workspace is installed, the retail-workspace-page-config.xml file needs to be edited to reflect the new application.

The file as supplied comes with all appropriate products configured, but the configurations of non-installed products have been "turned off". Therefore, when "turning on" a product, locate the appropriate entry, set "rendered" to "true", and enter the correct URL and parameters for the new application.

The entry consists of the main URL string plus one parameter named "config". The value of the config parameter is inserted by the installer. Somewhere in the installer property files there is a value for the properties "deploy.retail.product.rms.url" and "deploy.retail.product.rms.config".

For example, suppose RMS was installed on mycomputer.mycompany.com, port 7777, using a standard install and rms configured with the application name of "rms121sedevhpsso". If you were to access RMS directly from your browser, you would type in:

http://mycomputer.mycompany.com:7777/forms/frmservlet?config=rms121sedevhpsso

The entry in the retail-workspace-page-config.xml after installation would resemble the following:

```xml
<url>http://mycomputer.mycompany.com:7777/forms/frmservlet</url>
<parameters>
  <parameter name="config">
    <value>rms121sedevhpsso</value>
  </parameter>
</parameters>
```
Setting up Oracle Business Intelligence Publisher

RMS uses Oracle Business Intelligence (BI) Publisher for publishing reports. Prior to accessing reports in BI Publisher, you must create the BI Publisher JDBC connection for your database and verify that BI Publisher URLs are set correctly on your Oracle Application Server.

**Note:** Oracle Retail recommends that retailers access the reports with the Guest user with no password. This is so RMS can access the reports without authentication. Guest with no password is a default user in BIP.

**Setting up your JDBC Connection**

Use the following procedure to create your JDBC connection for either a RAC database or a thin connection to a database.

1. Access your BIP repository.
   
   /home/BIP/

2. Log into BI Publisher as the administrator.

3. Under Data Sources, select JDBC Connections.

4. Create the BIP JDBC Connection.
   - If you are using JDBC for a RAC database, verify that the Data Source Name is set to RMS13.
Example of a valid JDBC for a RAC database

- If you are using a thin connection to a database, the Connection String should look similar to the following:
  `jdbc:oracle:thin:@<DB_Server>:<DB_listener_port>:<ORACLE_SID>
  
  For example:
  `jdbc:oracle:thin:@mspdev38.us.oracle.com:1525:cs0110g`
Verifying BI Publisher URLs

The BI Publisher URLs are set during the installation of RMS. If the URLs are incorrect, edit the RMS.env file using a text editor to correct the errors.

Use the following procedure to verify that your BI Publisher URLs are correct on your Oracle Application Server.

1. From your Oracle Application Server, access your RMS.env file. Your RMS.env file can be found in the following location:
   $ORACLE_HOME/forms/server/<DIR>/RMS*.env

2. Verify that the following lines in the RMS.env file correspond to your JDBC connection values:
   ORACLE_RMS_REPORTS_HOST=http://<server>:<port>/
   ORACLE_RMS_REPORTS_SERVER=<context root of BIP app>
   ORACLE_RMS_RWSERVER=<user>/<application>/<version>/

   For example:
   ORACLE_RMS_REPORTS_HOST=http://mspdev9091vip.us.oracle.com:7777/
   ORACLE_RMS_REPORTS_SERVER=xmlpserver
   ORACLE_RMS_RWSERVER=/Guest/RMS/12.1int/

3. Verify that the "RMS13" datasource is set correctly in the xdo files.
   In all reports, there is a <report>.xdo. Those xdo files have a similar line that should be set to "RMS13" out of the box.
   <report version="1.1" xmlns="http://xmlns.oracle.com/oxp/xmlp"
   defaultDataSourceRef="RMS13">