Oracle® Retail Merchandising System Back-end Configuration and Operations Guide - Volume 3
Release 12.0
May 2006

ORACLE

Contents

Preface

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

Audience

Anyone with an interest in developing a deeper understanding of the underlying processes and architecture supporting RMS functionality will find valuable information in this guide. There are three audiences in general for whom this guide is written:

- Business analysts looking for information about processes and interfaces to validate the support for business scenarios within RMS and other systems across the enterprise.
- System analysts and system operations personnel who:
  - Seek information about the RMS processes internally or in relation to the systems across the enterprise.
  - Operate RMS regularly.
- Integrators and implementation staff with overall responsibility for implementing RMS.

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 Online Help
- Oracle Retail Merchandising System User Guide
- Oracle Retail Merchandising System Data Model
- Oracle Retail Merchandising System Batch Schedule
- Oracle Retail Merchandising System Operations Guide
  - Volume 1, Batch Overviews and Designs
  - Volume 2, Message Publication and Subscription Designs

Customer Support

- https://metalink.oracle.com

When contacting Customer Support, please provide:

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

1

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:

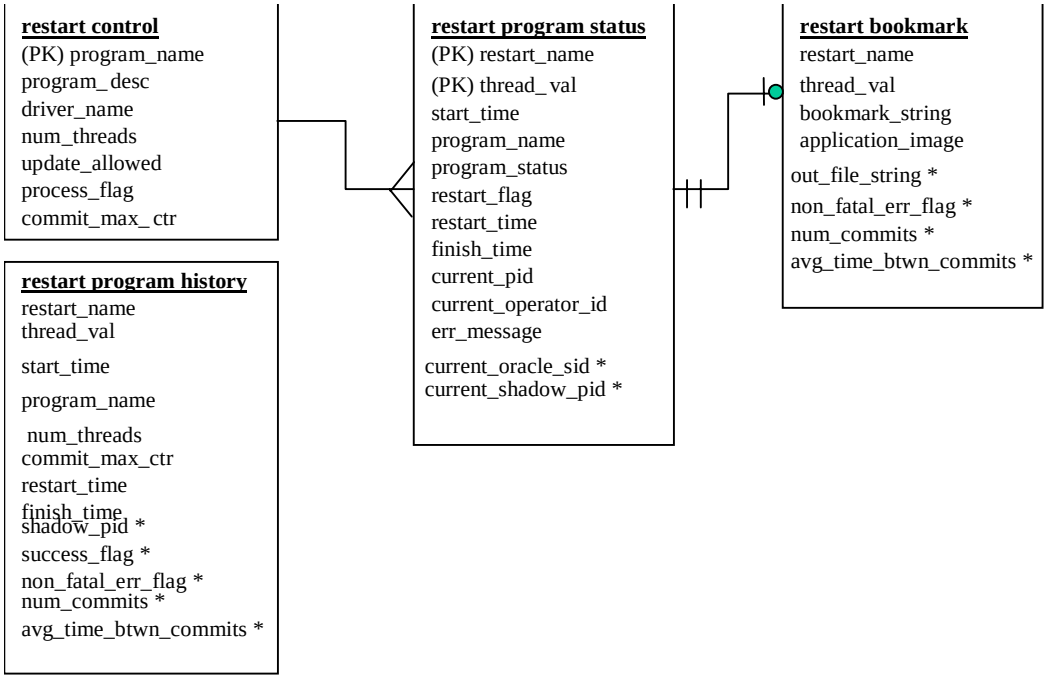
- Recover a halted process from the point of failure
- Prevent system halts due to large numbers of transactions
- Allow multiple instances of a given process to be active at the same time

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.



Entity Relationship

Note: The fields with asterisks (*) are only used by new batch programs of release 9.0 or later.

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.

restart_control			
(PK) program_name	varchar2	25	Batch program name
program_desc	varchar2	50	A brief description of the program function
driver_name	varchar2	25	Driver on query, for example, department (non-updatable)
num_threads	num	10	Number of threads used for current process
update_allowed	varchar2	2	Indicates whether user can update thread numbers or if done programmatically
process_flag	varchar2	1	Indicates whether process is table-based (T) or file-based (F)
commit_max_ctr	num	6	Numeric maximum value for counter before commit occurs

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).

restart_program_status			
(PK)restart_name	varchar2	50	Program name
(PK)thread_val	num	10	Thread counter
start_time	date		dd-mon-yy hh:mi:ss
program_name	varchar2	25	Program name
program_status	varchar2	25	Started, aborted, aborted in init, aborted in process, aborted in final, completed, ready for start
restart_flag	varchar2	1	Automatically set to 'N' after abnormal end, must be manually set to 'Y' for program to restart
restart_time	date		dd-mon-yy hh:mi:ss
finish_time	date		dd-mon-yy hh:mi:ss
current_pid	num	15	Starting program id
current_operator_id	varchar2	20	Operator that started the program
err_message	varchar2	255	Record that caused program abort & associated error message
current_oracle_sid	num	15	Oracle SID for the session associated with the current process
current_shadow_pid	num	15	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.

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.

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

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.

restart_bookmark			
(PK) restart_name	varchar2	50	Program name
(PK) thread_val	num	10	Thread counter
bookmark_string	varchar2	255	Character string of key of last committed record.
application_image	varchar2	1000	application parameters from the last save point.
out_file_string	varchar2	255	Concatenated file pointers (UNIX sometimes refers to these as stream positions) of all the output files from the last commit point of the current process. It is used to return to the right restart point for all the output files during restart process.
non_fatal_err_flag	varchar2	1	Indicates whether non-fatal errors have occurred for the current process.
num_commits	num	12	Number of commits for the current process. The possible last commit when restart/recovery is closed is not counted.
avg_time_btwn_commits	num	12	Average time between commits for the current process. The possible last commit when restart/recovery is closed is not counted.

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.

v_restart_x		
driver_name	varchar2	- example dept, store, region, etc.
num_threads	number	total number of threads in set (defined on restart control)
driver_value	number	- will be the numeric value of the driver_name
thread_val	number	thread value defined for driver_value and num_threads combination

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:

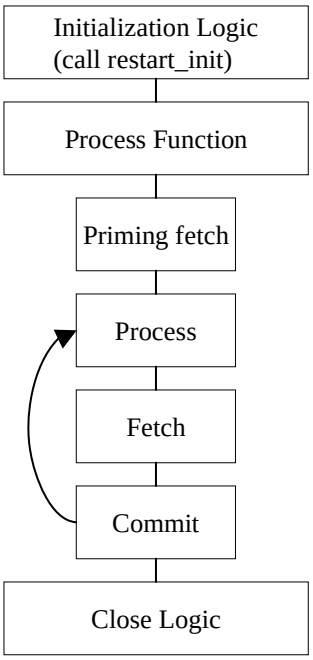
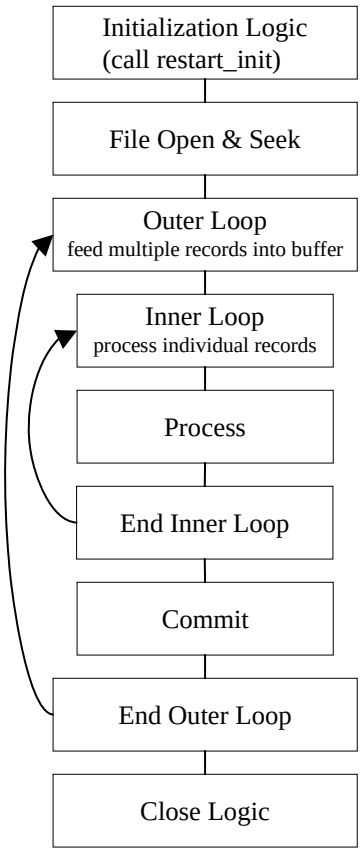


Table-Based Restart/Recovery Program Flow

The following diagram depicts *file*-based Restart/Recovery program flow



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/cursors
- 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

The current RMS restart/recovery library has been updated in RMS versions 9, 10, 11, and 12 to enhance maintainability, enable easier coding and improve performance. While the current mechanism and functionality of batch restart/recovery are preserved, the following improvements and enhancements have been done:

- Organize global variables associated with restart recovery
- Allow the batch developer full control of restart recovery variables parameter passing during initialization
- Remove temporary write files to speed up the commit process
- Move more information and processing from the batch code into the library code
- Add 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.

```
#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_curren_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_curren_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.

2
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:

```
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 bet 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 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.

Driver_name	num_threads	driver_val	thread_val
DEPT	1	101	1
DEPT	2	101	2
DEPT	3	101	3
DEPT	4	101	2
DEPT	5	101	2
DEPT	6	101	6
DEPT	7	101	4

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

Driver_name	num_threads	driver_val	thread_val
STORE	5	1	2
STORE	5	2	3
STORE	5	3	4
STORE	5	4	5
STORE	5	5	1
STORE	5	6	2
STORE	5	7	3
STORE	5	8	4
STORE	5	9	5
STORE	5	10	1

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):

```
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:

row #	restart_name	thread_val	program_name	etc…
1	SALDLY	1	SALDLY	…
2	SALDLY	2	SALDLY	…

restart_program_status table after insert:

row #	restart_name	thread_val	program_name	etc…
1	SALDLY	1	SALDLY	…
2	SALDLY	2	SALDLY	…
3	SALDLY	3	SALDLY	…

restart_program_status table after delete:

row #	restart_name	thread_val	program_name	etc…
1	SALDLY	1	SALDLY	…

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.

3
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.

4

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
FDETL0000000002SKU100000040000011011
FDETL0000000003SKU100000050003002001
FDETL0000000004SKU100000050003002001
FTAIL0000000050000000003
```

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:

```
FHEAD0000000001RTV 19960908172000
THEAD000000000200000000000000119960909120200000000000003R
TDETL000000000300000000000000001000001SKU10000012
TTAIL0000000004000001
THEAD000000000500000000000000219960909120200121572013 1R
TDETL000000000600000000000000002000001UPC400100002667
TDETL000000000700000000000000020000021UPC400100002643 0
TTAIL0000000008000002
FTAIL000000000900000000007
```

Record Name	Field Name	Field Type	Default Value	Description
File Header	File Type Record Descriptor	Char(5)	FHEAD	Identifies file record type
	File Line Identifier	Number(10)	Specified by external system	Line number of the current file
	File Type Definition	Char(4)	n/a	Identifies transaction type
	File Create Date	Date	Create date	Date file was written by external system
Transaction Header	File Type Record Descriptor	Char(5)	THEAD	Identifies file record type
	File Line Identifier	Number(10)	Specified by external system	Line number of the current file
	Transaction Set Control Number	Char(14)	Specified by external system	Used to force unique transaction check
	Transaction Date	Char(14)	Specified by external system	Date the transaction was created in external system

Record Name	Field Name	Field Type	Default Value	Description
Transaction Detail	File Type Record Descriptor	Char(5)	TDETL	Identifies file record type
	File Line Identifier	Number(10)	Specified by external system	Line number of the current file
	Transaction Set Control Number	Char(14)	Specified by external system	Used to force unique transaction check
	Detail Sequence Number	Char(6)	Specified by external system	Sequential number assigned to detail records within a transaction
Transaction Trailer	File Type Record Descriptor	Char(5)	TTAIL	Identifies file record type
	File Line Identifier	Number(10)	Specified by external system	Line number of the current file
	Transaction Detail Line Count	Number(6)	Sum of detail lines	Sum of the detail lines within a transaction
File Trailer	File Type Record Descriptor	Char(5)	FTAIL	Identifies file record type
	File Line Identifier	Number(10)	Specified by external system	Line number of the current file
	Total Transaction Line Count	Number(10)	Sum of all transaction lines	All lines in file less the file header and trailer records

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.

5

RETL Program Overview for RMS/ReSA Extractions

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.

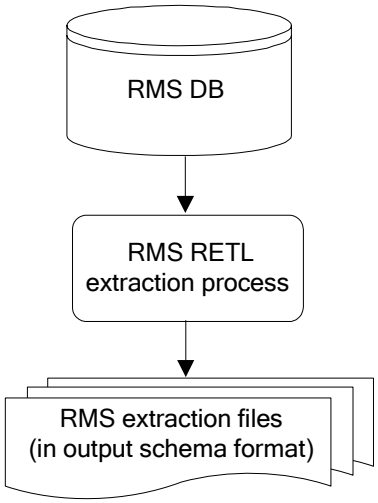
Architectural Design

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

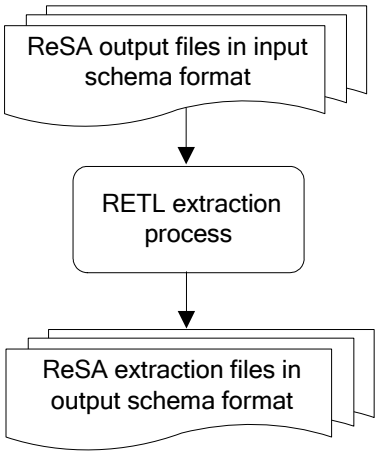
The architecture relies upon the use of well-defined flows specific to the RMS database. The resulting output is comprised of data files written in a well-defined schema file format. This extraction includes no destination specific code.



RETL Extraction Processing for RMS

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.



RETL Extraction Processing for ReSA

Configuration

RETL

Before trying to configure and run RMS ETL, install RETL version 12.0 or later, which is required to run RMS 12.0 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"
```

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:
   a. Change the DBNAME variable to the name of the RMS database.
   b. Change the RMS_OWNER variable to the username of the RMS schema owner.
   c. Change the BA_OWNER variable to the username of the RMSE batch user.
   d. Change the ORACLE_HOST variable to the database server name.
   e. Change the ORACLE_PORT variable to the database port number
   f. Change the MAX_NUM_COLS variable to modify the maximum number of columns from which RETL selects records.

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 `dwidev`. Run the profile and verify that the `MMUSER` and `PASSWORD` variables are set to the batch user, dwidev, 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.

6

RMS Internationalization and Localization

The technical infrastructure of RMS supports languages other than English. The software can efficiently handle multiple languages. Tables have been added to RMS to accommodate internationalization. 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 form 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.

7
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.