Oracle® Retail Merchandising System Operations Guide - Volume 2 Message Publication and Subscription Designs Release 12.0 May 2006

ORACLE

Contents

Preface

Oracle Retail Operations Guides are designed so that you can view and understand the application's 'behind-the-scenes' processing. This volume of the Oracle Retail Merchandising System (RMS) Operations Guide includes the following information:

- Publication designs which describe, on a technical level, how RMS publishes messages to the Oracle Retail Integration Bus (RIB).
- Subscription designs which describe, on a technical level, how RMS subscribes to messages from the RIB.

Audience

This operations guide is designed for System Analysts and Database Administrators who are looking for technical descriptions of data processes by functional area.

Related Documents

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

- Oracle Retail Merchandising System Operations Guide – Volume 1
- Oracle Retail Merchandising System Operations Guide – Volume 3
- Oracle Retail Merchandising System Batch Schedule
- 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

  Note: Refer to the data model for information on the SYSTEM_OPTIONS table. The SYSTEM_OPTIONS table contains significant retailer-defined parameters. This table is populated during installation of the system and must be maintained by the database administrator.

- Oracle Retail Integration Bus (RIB) documentation
- Other Oracle Retail product documentation

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
Publication Designs

Allocations

Functional Area

Allocations

Design Overview

RMS is responsible for communicating allocation information with external systems such as a warehouse management system (RWMS, for example). Allocations include context information at the header level. The context_type defines the business reason for the allocation, thus allowing users to distinguish one form of allocation from another. For example, when the context of an allocation is promotion (that is, when the allocation is being created to support an RPM promotion), the ID of the promotion being supported is attached to the allocation.

Allocation data can enter RMS through the following three ways:

- Through the Oracle Retail Allocation product
  These allocations are written to the ALLOC_HEADER and ALLOC_DETAIL tables in 'R'eserved or 'A'pproved status. Once a detail and a header message have been queued and approved, a message is published to the RIB. Detail modification messages for allocations are not sent to the RIB.
- Through the semi-automatic ordering option
  Via this replenishment method, allocations and orders are inserted into the ALLOC_HEADER and ALLOC_DETAIL tables in worksheet status to be manually approved. In order for allocation messages to be published to the RIB, the allocation must at least be in 'A'pproved status. Worksheet messages remain on the queue and combined until they are approved. Once approval occurs, one create message is published to the RIB.
- Through automatic replenishment allocations
  These allocations are initially set in worksheet status and are approved by the RPLAPPRV.PC batch program (Replenishment Approve). Only messages for approved allocations are published to the RIB.

Modified and deleted allocation information is also sent to the RIB. Allocation header modification messages will be sent if the status of the allocation is changed to 'C'losed. Allocation detail modification messages will be sent for those allocations that were created via replenishment. Delete messages are ignored at the detail level. A header delete message signifies that the complete allocation can be deleted.

Functionality Checklist

Description	RMS	RIB
RMS must publish Allocations information		
Create new Publisher	X	X

Form Impact

None

Business Object Records

Create the following record types in the RMSSUB_ALLOC package body:

```
TYPE rowid_TBL     is table of ROWID
INDEX BY BINARY_INTEGER;

TYPE alloc_details_pub_TBL is table of ALLOC_DETAILS_PUBLISHED.ALLOC_NO%TYPE
        INDEX BY BINARY_INTEGER;

TYPE alloc_details_pub_to_loc_TBL     is table of
ALLOC_DETAILS_PUBLISHED.TO_LOC_VIR%TYPE
        INDEX BY BINARY_INTEGER;
```

Package Impact

Create Header

1. Prerequisites: Allocation can be created in one of three manners: via the stand-alone allocations product, semi-automatic ordering, or automatic ordering replenishment.
2. Activity Detail: Once an allocation exists in RMS it can be modified or details can be attached.
3. Messages: When an allocation is created an “Allocation Create” message request is queued. The Allocation Created message is a flat message containing a full snapshot of the allocation at the time the message is published (asynchronously from the modification). The message will not be sent until detail records have been queued and the allocation has been approved.

Modify Header

1. Prerequisites: An allocation must exist before it can be modified.
2. Activity Detail: The user is allowed to change the status of the allocation to ‘A’pproved or ‘C’losed. This change is of interest to other systems and so this activity results in the publication a message. Messages are only written for changes created by replenishment.
3. Messages: When an allocation is modified, an “Allocation Header Modified message” request is queued. The Allocation Header Modified message is a flat message containing a full snapshot of the allocation header at the time the message is published (asynchronously from the modification).

Create Detail

1. Prerequisites: An allocation header must exist before and allocation detail can be created and it can be loaded into RMS. Once in RMS, the allocation can only be modified by having its allocated quantity changed.
2. Activity Detail: When an “Allocation Detail Create” message is queued it could be the first time systems external to Oracle Allocation and RMS might have any interest at all in the existence of the allocation, so this is the first part of the life cycle of an allocation that is published if a “Create Header” message is also on the queue for the same allocation.

3. Messages: When an allocation detail is created, an “Allocation Detail Created message” request is queued. The Allocation Create message is a flat message containing a full snapshot of the allocation at the time the message is published (asynchronously from the modification). If an Allocation Create message is also in the queue for the same allocation, the two messages are combined and sent as one message.

Modify Detail

1. Prerequisites: An allocation detail must exist to be modified.
2. Activity Detail: The user is allowed to change allocation quantities provided they are not reduced below those already recorded as received. This change is of interest to other systems and so this activity results in the publication of a message. Messages are only written for changes created by replenishment.
3. Messages: When an allocation is modified an “Allocation Detail Modified message” request is queued. The Allocation Detail Modified message is a flat message containing a full snapshot of the allocation detail at the time the message is published (asynchronously from the modification).

Approve

1. Prerequisites: An allocation must exist in RMS before it can be approved for replenishment allocations. Those direct from the allocations product can be entered into RMS in approved status.
2. Activity Detail: Once an allocation as been approved, it will be the first time systems external to Allocations and RMS might have any interest at all in the existence of the allocation, so this is the first part of the life cycle of an allocation that is published if a “Create Header” message is also on the queue for the same allocation.
3. Messages: When the allocation is approved an “Allocation Header Modification” message is queued. This message will be combined with any Allocation Create and Allocation Detail Create message to form the message that is sent to the RIB.

Close

1. Prerequisites: An allocation must be approved before it can be closed.
2. Activity Detail: Closing an allocation changes the status, which prevents further receiving or modification of the allocation. When an allocation is closed a message is published to update other systems regarding the status change.
3. Messages: Closing an allocation queues an “Allocation Header Modified message” request. This is a flat message containing a full snapshot of the allocation at the time that the message is published (asynchronously from the activity).

Delete

1. Prerequisites: An allocation can only be deleted when it is still in approved status or when it has been closed. Note that if the allocation is in closed status, it still cannot be deleted if either a create or modify message, which need to take full snapshots, are pending for the allocation.
2. Activity Detail: Deleting an allocation removes it from the system. External systems are notified by a published message.
3. Message: When an allocation is deleted an Allocation Header Deleted message, which is a flat notification message, is queued.

Package Name: RMSMFM_ALLOC

Body File Name: rmsmfm_allocb.pls

Package Specification – Global Variables

```
FAMILY              CONSTANT  VARCHAR2(30)    := 'alloc';

HDR_ADD             CONSTANT  VARCHAR2(30)    := 'AllocCre';
HDR_UPD             CONSTANT  VARCHAR2(30)    := 'AllocHdrMod';
HDR_DEL             CONSTANT  VARCHAR2(30)    := 'AllocDel';
DTL_ADD             CONSTANT  VARCHAR2(30)    := 'AllocDtlCre';
DTL_UPD             CONSTANT  VARCHAR2(30)    := 'AllocDtlMod';
DTL_DEL             CONSTANT  VARCHAR2(30)    := 'AllocDtlDel';
```

Function Level Description – ADDTOQ

```
FUNCTION ADDTOQ(O_error_msg           OUT    VARCHAR2,
                I_message_type        IN     ALLOC_MFQUEUE.MESSAGE_TYPE%TYPE,
                I_alloc_no            IN     ALLOC_HEADER.ALLOC_NO%TYPE,
                I_alloc_header_status IN     ALLOC_HEADER.STATUS%TYPE,
                I_to_loc              IN     ITEM_LOC.LOC%TYPE)
```

This function is called by both alloc_header trigger and alloc_detail trigger, ec_table_alh_aiudr and ec_table_ald_aiudr respectively.

- For header level insert messages (HDR_ADD), insert a record in the ALLOC_PUB_INFO table. The published flag should be set to ‘N’. The correct thread for the Business transaction should be calculated and written. Call API_LIBRARY.RIB_SETTINGS to get the number of threads used for the publisher. Using the number of threads, and the Business object id, calculate the thread value.
- For all records except header level inserts (HDR_ADD), the thread_no and initial_approval_ind should be queried from the ALLOC_PUB_INFO table.
- If the Business transaction has not been approved (initial_approval_ind = ‘N’) and the triggering message is one of DTL_ADD, DTL_UPD, DTL_DEL, HDR_DEL, no processing should take place and the function should exit.
- For detail level message deletes (DTL_DEL), the system only needs one (the most recent) record per detail in the ALLOC_MFQUEUE. Delete any previous records that exist on the ALLOC_MFQUEUE for the record that has been passed. If the publish_ind is ‘N’, do not add the DTL_DEL message to the queue.
- For detail level message updates (DTL_UPD), the system only needs one DTL_UPD (the most recent) record per detail in the ALLOC_MFQUEUE. Delete any previous DTL_UPD records that exist on the ALLOC_MFQUEUE for the record that has been passed.
- For header level delete messages (HDR_DEL), delete every record in the queue for that allocation.
- For header level update message (HDR_UPD), update the ALLOC_PUB_INFO.INITIAL_APPROVAL_IND to ‘Y’ if the allocation is in approved status.
- For all records except header level inserts (HDR_ADD), insert a record into the ALLOC_MFQUEUE.

It returns a status code of API_CODES.SUCCESS if successful, API_CODES.UNHANDLED_ERROR if not.

Function Level Description – GETNXT

```
PROCEDURE GETNXT(  O_status_code      OUT    VARCHAR2,
                   0_error_msg        OUT    VARCHAR2,
                   O_message_type     OUT    VARCHAR2,
                   O_message          OUT    RIB_OBJECT,
                   O_bus_obj_id       OUT    RIB_BUSOBJID_TBL,
                   O_routing_info     OUT    RIB_ROUTINGINFO_TBL,
                   I_num_threads      IN     NUMBER DEFAULT 1,
                   I_thread_val       IN     NUMBER DEFAULT 1)
```

The RIB calls GETNXT to get messages. It performs a cursor loop on the unpublished records on the ALLOC_MFQUEUE table (PUB_STATUS = 'U'). It should only need to execute one loop iteration in most cases. For each record retrieved, GETNXT gets the following:

1. A lock of the queue table for the current Business object. The lock is obtained by calling the function LOCK_THE_BLOCK. If there are any records on the queue for the current Business object that are already locked, the current message is skipped.
2. The published indicator from the ALLOC_PUB_INFO table.
3. A check for records on the queue with a status of 'H'ospital. If there are any such records for the current Business object, GETNXT raises an exception to send the current message to the Hospital.

The loop will need to execute more than one iteration for the following cases:

1. When a header delete message exists on the queue for a Business object that has not been initially published. In this case, simply remove the header delete message from the queue and loop again.
2. A detail delete message exists on the queue for a detail record that has not been initially published. In this case, simply remove the detail delete message from the queue and loop again.
3. The queue is locked for the current Business object

The information from the ALLOC_MFQUEUE and ALLOC_PUB_INFO table is passed to PROCESS_QUEUE_RECORD. PROCESS_QUEUE_RECORD will build the Oracle Object message to pass back to the RIB. If PROCESS_QUEUE_RECORD does not run successfully, GETNXT raises an exception.

If any exception is raised in GETNXT, including the exception raised by an unsuccessful call to PROCESS_QUEUE_RECORD, HANDLE_ERRORS is called.

Function Level Description – PUB_RETRY

PROCEDURE PUB_RETRY

Same as GETNXT except:

It only loops for a specific row in the ALLOC_MFQUEUE table. The record on ALLOC_MFQUEUE must match the passed in sequence number (contained in the ROUTING_INFO).

Function Level Description – PROCESS_QUEUE_RECORD (local)

This function controls the building of Oracle Objects given the Business transaction's key values and a message type. It contains all of the shared processing between GETNXT and PUB_RETRY.

Check to see if the Business object is being published for the first time. If the published_ind on the pub_info table is 'N', the Business object is being published for the first time. If so, call MAKE_CREATE.

Otherwise,

If the record from ALLOC_MFQUEUE table is a header update (HDR_UPD):

- Call BUILD_HEADER_OBJECT to build the Oracle Object to publish to the RIB.
- Update ALLOC_PUB_INFO with updated new header information
- Build the ROUTING_INFO.
- Delete the record from the ALLOC_MFQUEUE table.

If the record from ALLOC_MFQUEUE table is a detail add or update (DTL_ADD, DTL_UPD):

- Call BUILD_DETAIL_CHANGE_OBJECTS to build the Oracle Object to publish to the RIB. This will also take care of any ALLOC_MFQUEUE deletes and ROUTING_INFO logic.

If the record from ALLOC_MFQUEUE table is a detail delete (DTL_DEL):

- Call BUILD_DETAIL_DELETE_OBJECTS to build the Oracle Object to publish to the RIB. This will also take care of any ALLOC_MFQUEUE and ALLOC_DETAILS_PUBLISHED deletes and the ROUTING_INFO logic.

Function Level Description – MAKE_CREATE (local)

This function is used to create the Oracle Object for the initial publication of a Business transaction.

- Call BUILD_HEADER_OBJECT to get a header level Oracle Object plus any extra functional holders.
- Build some or all of the ROUTING_INFO Oracle Object.
- Call BUILD_DETAIL_OBJECTS to get a table of detail level Oracle objects and a table of ALLOC_MFQUEUE rowids to delete.
- Use the header level Oracle Object and functional holders to update the ALLOC_PUB_INFO.
- Delete records from the ALLOC_MFQUEUE for all rowids returned by BUILD_DETAIL_OBJECTS. Deletes are done by rowids instead of Business transaction keys to ensure that noting is deleted off the queue that has not been published.
- If the entire Business transaction was added to the Oracle Object, also delete the ALLOC_MFQUEUE record that was picked up by GETNXT. If the entire Business transaction was not published, the system needs to leave something on the ALLOC_MFQUEUE to ensure that the rest of it is picked up by the next call to GETNXT.
- The header and detail level Oracle Objects are combined and returned.

Function Level Description – BUILD_HEADER_OBJECT (local)

Accepts header key values, performs necessary lookups, builds and returns a header level Oracle Object.

Optionally can return needed Functional Holders for the ALLOC_PUB_INFO table.

The C_ALLOC_HEAD cursor is modified to select the context fields (context and value) off the alloc_header table.

The context fields will be passed along in the parameter list of the rib object constructor RIB_ALLOCDESC_REC().

Function Level Description – BUILD_DETAIL_OBJECTS (local)

The function is responsible for building detail level Oracle Objects. It builds as many detail Oracle Object as it can given the passed in message type and Business object keys.

If the function is being called from MAKE_CREATE:

Select any unpublished detail records from the Business transaction (use an indicator on the functional detail table itself or ALLOC_DETAILS_PUBLISHED). Create Oracle Objects for details that are selected by calling BUILD_SINGLE_DETAIL.

- Ensure that the indicator in the functional detail table is updated as published as the detail info are placed into the Oracle Objects
- Ensure that ALLOC_MFQUEUE is deleted as needed. If there is more than one ALLOC_MFQUEUE record for a detail level record, make sure they all get deleted. The system only cares about current state, not every change.
- Ensure that ROUTING_INFO is constructed if routing information is stored at the detail level in the Business transaction.
- Ensure that no more than MAX_DETAILS_TO_PUBLISH records are put into Oracle Objects.
- Ensure that the detail records being added to the object have not already been published. This can happen if GETNXT was previously called for the current Business object, and the MAX_DETAILS_TO_PUBLISH limit had been reached. The system ensures these details do not get added again by looking at the indicator in the functional detail table.

If the function is not being called from MAKE_CREATE:

Select any details on the ALLOC_MFQUEUE that are for the same Business transaction and for the same message type. Create Oracle Objects for details that are selected by calling BUILD_SINGLE_DETAIL.

- If the message type is a detail create (DTL_ADD), ensure that records get inserted into ALLOC_DETAILS_PUBLISHED or the indicator in the functional detail table is updated as published because the detail info are placed into the Oracle Objects.
- Ensure that ALLOC_MFQUEUE is deleted from as needed.
- Ensure that ROUTING_INFO is constructed if routing information is stored at the detail level in the Business transaction.
- Ensure that no more than MAX_DETAILS_TO_PUBLISH records are put into Oracle Objects.

The deletes are done by ROWID to make sure that records from the queue table that has not been published are not deleted.

Function Level Description – BUILD_SINGLE_DETAIL (local)

Accept inputs and build a detail level Oracle Object. Perform any lookups needed to complete the Oracle Object.

Function Level Description – BUILD_DETAIL_CHANGE_OBJECTS (local)

Either pass in a header level Oracle Object or call BUILD_HEADER_OBJECT to build one.

Call BUILD_SINGLE_DETAIL to get the detail level Oracle Objects.

Perform any BULK DML statements given the output from BUILD_ DETAIL_OBJECTS.

Build any ROUTING_INFO as needed.

Function Level Description – BUILD_DETAIL_DELETE_OBJECTS (local)

Either pass in a header level ref Oracle Object or build a header level ref Oracle Object.

Perform a cursor for loop on ALLOC_MFQUEUE and build as many detail ref Oracle Objects as possible without exceeding the MAX_DETAILS_TO_PUBLISH.

Perform any BULK DML statements for deletion from ALLOC_MFQUEUE and update to ALLOC_DETAILS_PUBLISHED.

Build any ROUTING_INFO as needed.

Function Level Description – LOCK_THE_BLOCK (local)

This function locks all queue records for the current Business object. This is to ensure that GETNXT does not wait on any business processes that currently have the queue table locked and have not committed. This can occur because ADDTOQ, which is called from the triggers, deletes from the queue table for DTL_UPD, DTL_DEL, and HDR_DEL messages.

Function Level Description – HANDLE_ERRORS (local)

HANDLE_ERRORS is called from GETNXT and PUB_RETRY when an exception is raised.

If the error is a non-fatal error, GETNXT passes the sequence number of the driving ALLOC_MFQUEUE record back to the RIB in the ROUTING_INFO. It sends back a status of ‘H’ospital to the RIB as well. It then updates the status of the queue record to ‘H’ospital, so that it will not get picked up again by the driving cursor in GETNXT.

If the error is a fatal error, a status of ‘E’rror is returned to the RIB.

The error is considered non-fatal if no DML has occurred yet. Whenever DML has occurred, then the global variable LP_error_status is flipped from ‘H’ospital to ‘E’rror.

Function Level Description – DELETE_QUEUE_REC (local)

This function deletes a specific record on ALLOC_MFQUEUE table depending on the seq_no.

Function Level Description – GET_ROUTING_TO_LOCS (local)

This function will get all the values of to_loc_vir from alloc_details_published table depending on a given alloc number.

Perform a cursor for loop that will populate the oracle object RIB_ROUTINGINFO_TBL.

Function Level Description – GET_NOT_BEFORE_DAYS (local)

This function will get data from code_detail table and checks if the variable (LP_nbf_days) has a value or not. If there is no value, it will populate the variable then assign this value to the variable O_days.

Function Level Description – GET_RETAIL (local)

This function will accept inputs and pass it to PRICING_ATTRIB_SQL.GET_RETAIL function to get the retail value of the item.

Function Level Description – CHECK_STATUS (local)

CHECK_STATUS raises an exception if the status code is set to 'E'rror. This should be called immediately after calling a procedure that sets the status code. Any procedure that calls CHECK_STATUS must have its own exception handling section.

Trigger Impact

Create a trigger on the ALLOC_HEADER table to capture Inserts, Updates, and Deletes.

Trigger Name: EC_TABLE_ALD_AIUDR

Trigger File Name: ec_table_ald_aiudr.trg

Table: ALLOC_DETAIL

Inserts

- Send the ALLOC detail level info to the ADDTOQ procedure in the MFM with the message type RMSMFM_ALLOC.DTL_ADD and the original message.

Updates

- Send the ALLOC detail level info to the ADDTOQ procedure in the MFM with the message type RMSMFM_ALLOC.DTL_UPD and the original message.

Deletes

- Send the ALLOC detail level info to the ADDTOQ procedure in the MFM with the message type RMSMFM_ALLOC.DTL_DEL and the original message.

DTD

Here are the filenames that correspond with each message type. Please consult RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
AllocCre	Allocation Create Message	AllocDesc.dtd

Message Types	Message Type Description	Document Type Definition (DTD)
AllocHdrMod	Allocation Header Modify Message	AllocHdrDesc.dtd
AllocDel	Allocation Delete Message	AllocRef.dtd
AllocDtlCre	Allocation Detail Create Message	AllocDtlDesc.dtd
AllocDtlMod	Allocation Detail Modify Message	AllocDtlDesc.dtd
AllocDtlDel	Allocation Detail Delete Message	AllocDtlRef.dtd

Table Impact

TABLE	SELECT	INSERT	UPDATE	DELETE
ALLOC_PUB_INFO	Yes	Yes	Yes	No
ALLOC_MFQUEUE	Yes	Yes	No	Yes
ALLOC_DETAILS_PUBLISHED	Yes	Yes	Yes	Yes
ALLOC_HEADER	Yes	No	No	No
ALLOC_DETAIL	Yes	No	No	No
ITEM_MASTER	Yes	No	No	No
ITEM_TICKET	Yes	No	No	No
ITEM_LOC	Yes	No	No	No
WH	Yes	No	No	No
ORDHEAD	Yes	No	No	No
CODE_DETAIL	Yes	No	No	No
V_PACKSKU_QTY	Yes	No	No	No

Design Assumptions

- One of the primary assumptions in the current approach is that ease of code will outweigh performance considerations. It is hoped that the ‘trickle’ nature of the flow of data will decrease the need to dwell on performance issues and instead allow developers to code in the easiest and most straight forward manner.
- The adaptor is only set up to call stored procedures, not stored functions. Any public program then needs to be a procedure.

Banner

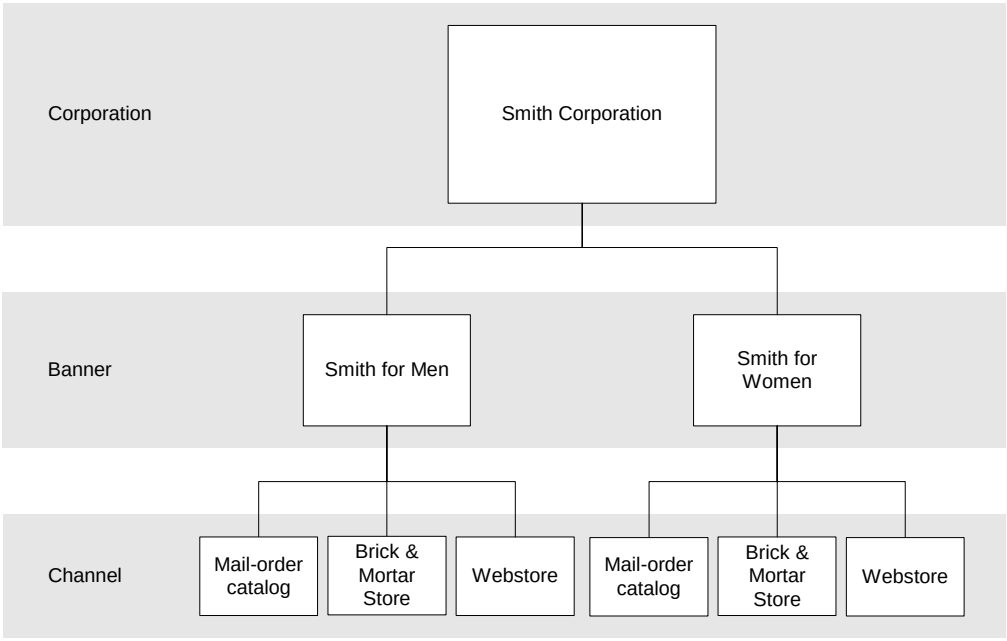
Business Overview

RMS publishes messages about banners and channels to the Oracle Retail Integration Bus (RIB). A banner provides a means of grouping channels thereby allowing the customer to link all brick and mortar stores, catalogs, and web stores. The BANNER table holds a banner identifier and name. The CHANNELS table shows all channels and any associated banner identifiers. In order to take advantage of banners and channels, the retailer must run RMS in a multi-channel environment.

> **Note:** To determine if your implementation of RMS is set up to run a multi-channel environment, look at the SYSTEM_OPTIONS table's multichannel_ind column for the value of 'Y' (Yes). If the multichannel_ind column's value is 'N' (No), multi-channel is not enabled.

For more information about multi-channels, see the chapter "Organization Hierarchy Batch" in volume 1 of this RMS Operations Guide.

The following diagram shows the structure of banners and channels within corporations.



Banners and channels within a corporation

Banner/channel publication consists of a single flat message containing information from the tables BANNER and CHANNELS. One message will be synchronously created and placed in the message queue each time a record is created, modified, or deleted. When a record is created or modified, the flat message will contain several attributes of the banner/channel. When a record is deleted, the message will simply contain the unique identifier of the banner/channel. Messages are retrieved from the message queue in the order they were created.

Functionality Checklist

Description	RMS	RIB
RMS must publish Banner/Channel information		
Create new Publisher	X	X

Form Impact

None.

Business Object Records

None

Package Impact

Create

1. Prerequisites: For channel creation, the associated banner must have been created.
2. Activity Detail: Once a banner/channel has been created, it is ready to be published. An initial publication message is made.
3. Messages: A “Banner Create” / “Channel Create” message is queued. This message is a flat message that contains a full snapshot of the attributes on the BANNER or CHANNEL table.

Modify

1. Prerequisites: banner/channel has been created.
2. Activity Detail: The user is allowed to change attributes of the banner/channel. These changes are of interest to other systems and so this activity results in the publication of a message.
3. Messages: Any modifications will cause a “banner modify” / channel modify” message to be queued. This message contains the same attributes as the “banner create” / “channel create” message.

Delete

1. Prerequisites: banner/channel has been created.
2. Activity Detail: Deleting a banner/channel removes it from the system. External systems are notified by a published message.
3. Messages: When a banner/channel is deleted, a “Banner Delete” / “Channel Delete” message, which is a flat notification message, is queued. The message contains the banner/channel identifier.

Package Name: RMSMFM_banner

Spec File Name: rmsmfm_banners.pls

Body File Name: rmsmfm_bannerb.pls

Package Specification – Global Variables

None

Function Level Description – ADDTOQ

```
Procedure: ADDTOQ
           (O_status          OUT          VARCHAR2,
            O_text            OUT          VARCHAR2,
            I_message_type    IN           BANNER_MFQUEUE.MESSAGE_TYPE%TYPE,
            I_banner_id       IN           CHANNELS.BANNER_id%TYPE,
            I_channel_id      IN           CHANNELS.CHANNEL_ID%TYPE,
            I_message         IN           CLOB)
```

This procedure is called by EC_TABLE_BAN_AIUDR and EC_TABLE_CHN_AIUDR, and takes the message type, banner ID, channel ID (NULL if called from EC_TABLE_BAN_AIUDR) and the message itself. It inserts a row into the message family queue BANNER_MFQUEUE along with the passed in values and the next sequence number from the message family sequence, setting the status to unpublished. It returns a status code of API_CODES.SUCCESS if successful, API_CODES.UNHANDLED_ERROR if not.

Function Level Description – GETNXT

```
Procedure: GETNXT
              (O_STATUS_CODE         OUT    VARCHAR2,
               O_ERROR_MSG           OUT    VARCHAR2,
               O_MESSAGE_TYPE        OUT    VARCHAR2,
               O_MESSAGE             OUT    CLOB,
               O_banner_id           OUT    NUMBER,
               O_channel_id          OUT    NUMBER)
```

This publicly exposed procedure is typically called by a RIB publication adaptor. Its parameters are well defined and arranged in a specific order. The message type is the RIB defined short message name. The message is the XML message. The family key consists of the banner ID, which will be populated for all message types, and the channel ID, which can be NULL.

The error text parameter contains application-generated information, such as the application's sequence number of the message that failed, and the Oracle or other error that occurred when the retrieval failed.

Function Level Description – GETNXT(local)

This procedure fetches the row from the message queue table that has the lowest sequence number. The message is retrieved, and then the row is removed from the queue.

Trigger Impact

Create a trigger on the banner and channels tables to capture inserts, updates, and deletes.

Trigger Name: EC_TABLE_BAN_AIUDR.TRG

Trigger File Name: ec_table_ban_aiudr.trg

Table: BANNER

This trigger will capture inserts/updates/deletes to the BANNER table and write data into the BANNER_MFQUEUE message queue. It will call BANNER_XML.BUILD_MESSAGE to create the XML message, and then call RMSMFM_BANNER.ADDTOQ to insert this message into the message queue.

Inserts

- Send banner info to the ADDTOQ procedure in the MFM with the message type RMSMFM_FAMILY.BannerDesc and the original message.

Updates

- Send banner info to the ADDTOQ procedure in the MFM with the message type RMSMFM_FAMILY.BannerDesc and the original message

Deletes

- Send banner info to the ADDTOQ procedure in the MFM with the message type RMSMFM_FAMILY.BannerRef and the original message.

Trigger Name: EC_TABLE_CHN_AIUDR.TRG

Trigger File Name: ec_table_chn_aiudr.trg

Table: CHANNELS

This trigger will capture inserts/updates/deletes to the CHANNELS table and write data into the BANNER_MFQUEUE message queue. It will call CHANNEL_XML.BUILD_MESSAGE to create the XML message and then call RMSMFM_BANNER.ADDTOQ to insert this message into the message queue.

Inserts

- Send channel info to the ADDTOQ procedure in the MFM with the message type RMSMFM_FAMILY.ChannelDesc and the original message

Updates

- Send channel info to the ADDTOQ procedure in the MFM with the message type RMSMFM_FAMILY.ChannelDesc and the original message.

Deletes

- Send channel info to the ADDTOQ procedure in the MFM with the message type RMSMFM_FAMILY.ChannelRef and the original message.

DTD

Here are the filenames that correspond with each message type. Please consult the RIB documentation for each message type, in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
BannerCre	Banner Create Message	BannerDesc.dtd
BannerMod	Banner Modify Message	BannerDesc.dtd
BannerDel	Banner Delete Message	BannerRef.dtd
ChannelsCre	Channels Create Message	ChannelDesc.dtd
ChannelsMod	Channels Modify Message	ChannelDesc.dtd
ChannelsDel	Channels Delete Message	ChannelRef.dtd

Table Impact

TABLE	SELECT	INSERT	UPDATE	DELETE
BANNER_MFQUEUE	Yes	Yes	No	Yes

Design Assumptions

- One of the primary assumptions in the current approach is that ease of code will outweigh performance considerations. It is hoped that the ‘trickle’ nature of the flow of data will decrease the need to dwell on performance issues and instead allow developers to code in the easiest and most straight forward manner.
- The adaptor is only setup to call stored procedures, not stored functions. Any public program then needs to be a procedure.

Differentiator Groups

Business Overview

Differential group publication consists of a single flat message containing differential group attributes from the tables DIFF_GROUP_HEAD and DIFF_GROUP_DETAIL. One message will be synchronously created and placed in the message queue each time a differential group (DIFF_GROUP_HEAD) is created, modified, or deleted or when a differentiator (DIFF_GROUP_DETAIL) is created, modified, or deleted from a differential group. When a differential group (DIFF_GROUP_HEAD) is created or modified, the flat message will contain numerous attributes of the group. When a differential group is deleted, the message will simply contain the unique identifier of the group, diff_group_id. When a differentiator (diff_group_detail) is created or modified, the flat message will contain numerous attributes of the differentiator. When a differentiator is deleted, the message will simply contain the unique identifier of the differential group and the differentiator, diff_group_id and diff_id. Messages are retrieved from the message queue in the order they were created.

Functionality Checklist

Description	RMS	RIB
RMS must publish Differentiator Groups information		
Create new Publisher	X	X

Form Impact

None.

Business Object Records

None.

Package Impact

Create Diff_Group

1. **Prerequisites**: Diff_Group does not already exist.
2. **Activity Detail**: Any change to the DIFF_GROUP_HEAD table inserts a DiffGrpHdrCre message_type record on the DIFFGRP_MFQUEUE table.
3. **Messages**: The DiffGrpHdrDesc message is created. It is a flat, synchronous message containing a full snapshot of the diff_group at the time the message is published.

Modify Diff_Group

1. **Prerequisites**: Diff_Group exists.
2. **Activity Detail**: Any change to the DIFF_GROUP_HEAD table inserts a DiffGrpHdrMod message_type record on the DIFFGRP_MFQUEUE table.
3. **Messages**: The DiffGrpHdrDesc message is created. It is a flat, synchronous message containing a full snapshot of the diff_group at the time the message is published.

Create Diff_Group_Detail

1. **Prerequisites**: A Diff_Group already exists, and the diff_id exists on diff_ids, but the diff_id does not exist within the diff_group.
2. **Activity Detail**: Any Differentiators added to a diff_group inserts a record to the DIFF_GROUP_HEAD table. A DiffGrpDtlCre message type record is also inserted on the DIFFGRP_MFQUEUE table. A foreign key to the DIFF_GROUP_HEAD table checks the existence of the diff_group the value is created to supplement.
3. **Messages**: DiffGrpDtlDesc message type is created. It is a hierarchical, synchronous message containing a snapshot of the Diff_Group_Detail table at the time the message is published.

Modify Diff_Group_Detail

1. **Prerequisites**: Diff_Group and the Diff_id within the diff_group (diff_group_detail record) exist.
2. **Activity Detail**: Any change to the Differentiators within a diff_group modifies a record to the DIFF_GROUP_HEAD table. A DiffGrpDtlMod message type record is also inserted on the DIFFGRP_MFQUEUE table. A foreign key to the Diff_Group_Head table checks the existence of the diff_group the value is created to supplement
3. **Messages** DiffGrpDtlDesc message is created. It is a flat, synchronous message containing a snapshot of the DIFF_GROUP_DETAIL table at the time the message is published.

Delete Diff_Group_Detail

1. **Prerequisites**: Diff_Group and the Diff_id within the diff_group (diff_group_detail record) exist.
2. **Activity Detail**: Deleting a Differentiator from a Diff_Group removes it from the DIFF_GROUP_DETAIL table and inserts a DiffGrpDtlDel row to the DIFFGRP_MFQUEUE table.
3. **Message**: A DiffGrpDtlRef message is created. It is a flat, synchronous message containing the primary key with which the external systems can remove it from their systems.

Delete Diff_Group

1. **Prerequisites**: Diff_Group exists and a diff_id within the diff_group (diff_group_detail record) may or may not exist.
2. **Activity Detail**: Deleting a Diff_Group removes it from the DIFF_GROUP_HEAD table and inserts a DiffGrpDel row to the DIFFGRP_MFQUEUE table. Because the differentiator_group_maintenance.fmb form in RMS automatically removes any child records on the DIFF_GROUP_DETAIL table when the diff_group is removed, there will be a row inserted to the DIFFGRP_MFQUEUE table for each diff_group_detail record associated with the deleted diff_group as well. These will receive the lower sequence numbers so that these will be acted upon first in the message queue. They will look like the DELETE DIFF_GROUP_DETAIL message detailed in the section above.
3. **Message**: A DiffGrpRef message is created for the diff_group only. It is a flat, synchronous message containing the primary key with which the external systems can remove it from their systems.

Package Name: RMSMFM_DIFFGRP

Spec File Name: rmsmfm_diffgrps.pls

Body File Name: rmsmfm_diffgrpb.pls

Function Level Description – ADDTOQ

```
Function: ADDTOQ
           (O_status          OUT            VARCHAR2,
            O_text            OUT            VARCHAR2,
            I_message_type    IN             DIFFGRP_MFQUEUE.MESSAGE_TYPE%TYPE
            I_diff_group_id   IN             DIFFGRP_MFQUEUE.DIFF_GROUP_ID%TYPE,
            I_diff_id         IN             DIFFGRP_MFQUEUE.DIFF_ID%TYPE,
            I_message         IN             CLOB);
```

This procedure is called by an event capture trigger, and takes the message type, family key values and, for synchronously captured messages, the message itself. It inserts a row into the message family queue along with the passed in values and the next sequence number from the message family sequence, setting the status to unpublished, or skip in the case of consolidation messages. It returns error codes and strings according to the standards of the application in which it is being implemented.

Function Level Description – GETNXT

```
Procedure: GETNXT
           (O_status_code     OUT     VARCHAR2,
            O_error_msg       OUT     VARCHAR2,
            O_message_type    OUT     VARCHAR2,
            Message           OUT     CLOB,
            O_diff_group_id   OUT     DIFFGRP_MFQUEUE.DIFF_GROUP_ID%TYPE,
            O_diff_id         OUT     DIFFGRP_MFQUEUE.DIFF_ID%TYPE);
```

This publicly exposed procedure is typically called by a RIB publication adaptor. Its parameters are well defined and arranged in a specific order. The message type is the RIB defined short message name, the message is the xml message, and the family key(s) are the key for the message as pertains to the family, not all of which will necessarily be populated for all message types. Status code is one of five values, as shown in the following table.

The error text parameter contains application-generated information, such as the application's sequence number of the message that failed, and the Oracle or other error that occurred when the retrieval failed. The facility ID is only included in messages coming from RWMS.

Trigger Impact

Create a trigger on the DIFF_GROUP_HEAD and DIFF_GROUP_DETAIL table to capture inserts, updates, and deletes.

Trigger Name: EC_TABLE_DGH_AIUDR.TRG

Trigger File Name: ec_table_dgh_aiudr.trg

Table: Diff_Group_Head

Inserts

- Send the I_diff_group_id to the ADDTOQ procedure in the MFM with the message type RMSMFM_FAMILY.

Updates

- Send the I_diff_group_id to the ADDTOQ procedure in the MFM with the message type RMSMFM_FAMILY.
- Any change to the Diff_Group_Head table inserts a DiffGrpHdrCre message_type record on the DIFFGRP_MFQUEUE table

Deletes

- Send the I_diff_group_id to the ADDTOQ procedure in the MFM with the message type RMSMFM_FAMILY.

Trigger Name: EC_TABLE_DGD_AIUDR.TRG

Trigger File Name: ec_table_dgd_aiudr.trg

Table: Diff_Group_Detail

Inserts

- Send the I_diff_group_id, I_diff_id to the ADDTOQ procedure in the MFM with the message type RMSMFM_FAMILY.

Updates

- Send the I_diff_group_id, I_diff_id to the ADDTOQ procedure in the MFM with the message type RMSMFM_FAMILY.
- Any Differentiators added to a diff_group inserts a record to the DIFF_GROUP_HEAD table. A DiffGrpDtlCre message type record is also inserted on the DIFFGRP_MFQUEUE table. A foreign key to the DIFF_GROUP_HEAD table checks the existence of the diff_group the value is created to supplement.

Deletes

- Send the I_diff_group_id, I_diff_id to the ADDTOQ procedure in the MFM with the message type RMSMFM_FAMILY.

DTD

Here are the filenames that correspond with each message type. Please consult the RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
DiffGrpHdrCre	Differentiator Header Create Message	DiffGrpHdrDesc.dtd
DiffGrpHdrMod	Differentiator Header Modify Message	DiffGrpHdrDesc.dtd
DiffGrpHdrDel	Differentiator Header Delete Message	DiffGrpHdrRef.dtd
DiffGrpDtlCre	Differentiator Detail Create Message	DiffGrpDtlDesc.dtd
DiffGrpDtlMod	Differentiator Detail Modify Message	DiffGrpDtlDesc.dtd
DiffGrpDtlDel	Differentiator Detail Delete Message	DiffGrpDtlRef.dtd

Table Impact

TABLE	SELECT	INSERT	UPDATE	DELETE
DIFFGRP_MFQUEUE	YES	YES	NO	YES

Design Assumptions

- It is not possible for a detail trigger to accurately know the status of a header table.
- In order for the detail triggers to accurately know when to add a message to the queue, RMS should not allow approval of a business object while detail modifications are being made.
- It is not possible for a header trigger or a detail trigger to know the status of anything modified by GETNXT. If a header trigger or detail trigger is trying to delete queue records that GETNXT currently has locked, it will have to wait until GETNXT is finished and removes the lock. It is assumed that this time will be fairly short (at most 2-3 seconds.) It is also assumed that this will occur rarely because it involves updating/deleting detail records on a business object that has already been approved. This also has to occur at the same time GETNXT is processing the current business object.
- Delay all DML statements to as late a time as possible. Once DML statements have taken place, any error becomes a fatal error rather than a hospital error.

Differentiator ID

Business Overview

RMS publishes messages for diff identifiers (diff IDs), and diff groups.

When differentiators are created in RMS and need to be sent to other systems, they are sent out via differentiator ID publication. When the external system receives information about an item that includes the new diff ID, that system understands what the diff ID refers to.

For a general discussion of differentiators, see the section 'Diff Group' in the chapter "Subscription Design" in this volume of the RMS Operations Guide.

Diff Publication Concepts

Whenever RMS publishes item messages to the RIB, it can include all four diffs and their types. For more information about RMS item message publication, see the section 'Item' in the chapter "Publication Design" in this volume of the RMS Operations Guide.

Diff Message Processes

Diff message publication processes begin whenever a trigger 'fires' on one of the diff tables. When that occurs, the trigger extracts the affected row on the table and publishes the data to the corresponding message family queue staging table. A total of nine messages can be published; however, they group into these three categories:

- Group Header
- Group Details
- Diff IDs

Differentiator ID publication consists of a single flat message containing differentiator attributes from the table DIFF_IDS. One message will be synchronously created and placed in the message queue each time a differentiator (diff_ids) is created, modified, or deleted. When a differentiator (diff_ids) is created or modified, the flat message will contain numerous attributes of the differentiator. When a differentiator is deleted, the message will simply contain the unique identifier of the differentiator, diff_id. Messages are retrieved from the message queue in the order they were created.

Functionality Checklist

Description	RMS	RIB
RMS must publish Differentiator ID information		
Create new Publisher	X	X

Package Impact

Create Diff_Id

1. Prerequisites: Diff_Id does not already exist.
2. Activity Detail: Any change to the Diff_Ids table inserts a DiffCre message_type record on the DIFFID_MFQUEUE table.
3. Messages: The DiffDesc message is created. It is a flat, synchronous message containing a full snapshot of the diff_id at the time the message is published.

Modify Diff_Id

1. Prerequisites: Diff_Id exists.
2. Activity Detail: Any change to the DIFF_IDS table inserts a DiffMod message_type record on the DIFFID_MFQUEUE table.
3. Messages: The DiffDesc message is created. It is a flat, synchronous message containing a full snapshot of the diff_id at the time the message is published.

Delete Diff_Id

1. Prerequisites: Diff_Id exists.
2. Activity Detail: Deleting a Diff_Id removes it from the diff_ids table and inserts a DiffDel row to the DIFFID_MFQUEUE table.
3. Message: A DiffRef message is created. It is a flat, synchronous message containing the primary key with which the external systems can remove it from their systems.

Package Name: RMSMFM_DIFFID

Spec File Name: rmsmfm_diffids.pls

Body File Name: rmsmfm_diffidb.pls

Package Specification – Global Variables

None

Function Level Description – ADDTOQ

```
Function: ADDTOQ
          (O_status       OUT        VARCHAR2,
           O_text         OUT        VARCHAR2,
           I_message_type IN         DIFFID_MFQUEUE.MESSAGE_TYPE%TYPE,
           I_diff_id      IN         DIFFID_MFQUEUE.DIFF_ID%TYPE,
           I_message      IN         CLOB)
```

This procedure called by EC_TABLE_DID_AIUDR, takes the message type, diff_id, and the message itself. It inserts a row into the message family queue DIFFID_MFQUEUE along with the passed in values and the next sequence number from the message family sequence, setting the status to unpublished. It returns a status code of API_CODES.SUCCESS if successful, API_CODES.UNHANDLED_ERROR if not.

Function Level Description – GETNXT

```
Procedure: GETNXT
          (O_status_codeo     OUT     VARCHAR2,
           O_error_msg        OUT     VARCHAR2,
           O_message_type     OUT     VARCHAR2,
           O_message          OUT     CLOB,
           O_diff_id          OUT     DIFFGRP_MFQUEUE.DIFF_ID%TYPE)
```

This publicly exposed procedure is typically called by a RIB publication adaptor. Its parameters are well defined and arranged in a specific order. The message type is the RIB defined short message name, the message is the xml message, and the family key(s) are the key for the message as pertains to the family, not all of which will necessarily be populated for all message types.

The error text parameter contains application-generated information, such as the application's sequence number of the message that failed, and the Oracle or other error that occurred when the retrieval failed.

Function Level Description – GETNXT(local)

This procedure fetches the row from the message queue table that has the lowest sequence number. The message is retrieved and then the row is removed from the queue.

Trigger Impact

Create a trigger on the DIFF_IDS and DIFFID_MFQUEUE tables to capture Inserts, Updates, and Deletes.

Trigger Name: EC_TABLE_DID_AIUDR.TRG

Trigger File Name: ec_table_did_aiudr.trg

Table: DIFF_IDs

DIFFID_XML. BUILD_MESSAGE (O_status, O_text, O_message, I_record, I_action_type) – This function is called by the trigger EC_TABLE_DID_AIUDR on insert, update and delete of the DIFF_IDS table. This function gathers all the data necessary to build the message that needs to be sent to the Oracle Retail Integration Bus. It determines the proper message to build based on the action_type that is sent in the trigger. It builds DiffRef xml messages for delete statements, or DiffDesc xml messages for updates or inserts.

Inserts

- Sets action_type to 'A'dd and message_type to 'DiffCre'.

Updates

- Sets action_type to 'M'odify and message_type to 'DiffMod'..

Deletes

- Sets action_type to 'D'elete and message_type to 'DiffDel'.

DTD

Here are the filenames that correspond with each message type. Please consult the RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
DiffCre	Diffid Create Message	DiffDesc.dtd
DiffMod	Diffid Modify Message	DiffDesc.dtd
DiffDel	Diffid Delete Message	DiffRef.dtd

Table Impact

TABLE	SELECT	INSERT	UPDATE	DELETE
DIFFID_MFQUEUE	Yes	Yes	No	Yes

Design Assumptions

- One of the primary assumptions in the current approach is that ease of code will outweigh performance considerations. It is hoped that the ‘trickle’ nature of the flow of data will decrease the need to dwell on performance issues and instead allow developers to code in the easiest and most straight forward manner.
- The adaptor is only setup to call stored procedures, not stored functions. Any public program then needs to be a procedure.

Item

Business Overview

RMS publishes messages about items to the Oracle Retail Integration Bus (RIB). In situations where a retailer creates a new item in RMS, the message that ultimately is published to the RIB contains a hierarchical structure of the item itself along with all components that are associated with that item. Items and item components make up what is called the Items message family.

After the item creation message has been published to the RIB for use by external applications, any modifications to the basic item or its components cause the publication of individual messages specific to that component. Deletion of an item and component records has similar effects on the message modification process, with the exception that the delete message holds only the key(s) for the record.

Deposit Items

A deposit item is a product that has a portion which is returnable to the supplier and sold to the customer, with a deposit taken for the returnable portion. Because the contents portion of the item and the container portion of the item have to be managed in separate financial accounts (as the container item would be posted to a liabilities account) with different attributes, the retailer must set up two separate items. All returns of used deposit items (the returned item) are managed as a separate product, to track these products separately and as a generic item not linked to the actual deposit item (for example, bottles being washed and having no label).

The retailer can never put a container item on a transfer. Instead, the container item is added to returns to vendors (RTVs) automatically when the retailer adds the associated content item.

Deposit item attributes in RMS enable contents, container and crate items to be distinguished from one another. Additionally, it is possible to link a contents item to a container item for the purposes of inventory management.

In addition to contents and container items, many deposit items are delivered in plastic crates, which are also given to the customer on a deposit basis. These crates are sold to a customer as an additional separate product. Individual crates are not linked with contents or container items. Crates are specified in the system with a deposit item attribute.

From a receiving perspective, only the content item can be received. The receipt of a PO shows the container item but the receipt of a transfer does not. Similar to RTV functionality, online purchase order functionality automatically adds the container. The system automatically replicates all transactions for the container item in the stock ledger. In sum, for POs and RTVs, the container item is included; for transfers, no replication occurs.

Catch-weight Items

Retailers can order and manage products for the following types of catch-weight item:

- Type 1 – Purchase in fixed weight simple packs: sell by variable weight (for example, bananas)
- Type 2 – Purchase in variable weight simple packs: sell by variable weight (for example, ham on the bone sold on a delicatessen counter)
- Type 3 – Purchase in fixed weight simple packs containing a fixed number of eaches: sell by variable weight eaches (for example, pre-packaged cheese)
- Type 4 – Purchase in variable weight simple packs containing a fixed number of eaches: sell by variable weight eaches (for example, pre-packaged sirloin steak)

> **Note:** Oracle Retail suggests that catch-weight item cases be managed through the standard simple pack functionality.

In order for catch-weight items to be managed in RMS, the following item attributes are available:

- Cost UOM – All items in RMS will be able to have the cost of the item managed in a separate unit of measure (UOM) from the standard UOM. Where this is in a different UOM class from the standard UOM, case dimensions must be set up.
- Catch-weight item pack details – Tolerance values and average case weights are stored for catch-weight item cases to allow the retailer to report on the sizes of cases received from suppliers.
- Maximum catch-weight tolerance threshold
- Minimum catch-weight tolerance threshold

Retailers can set up the following properties for a catch-weight item:

- Order type
- Sale type

Retailers can also specify the following, at the item-supplier-country level:

- Cost unit of measure (CUOM)

Receiving and Inventory Movement Impact on Catch-weight Items

Inventory transaction messages include purchase order receiving, stock order receiving, returns to vendor, direct store delivery receiving, inventory adjustments and bill of lading. These messages include attributes that represent, for catch-weight items, the actual weight of goods involved in a transaction. These attributes are weight and weight UOM.

When RMS subscribes to inventory transaction messages containing such weight data, the transaction weight will be used for two purposes:

- To update weighted average cost (WAC) using the weight rather than the number of units

  and

- To update the average weight value of simple packs

> **Note:** The WAC calculation does not apply to return to vendors (RTVs).

Item Transformation

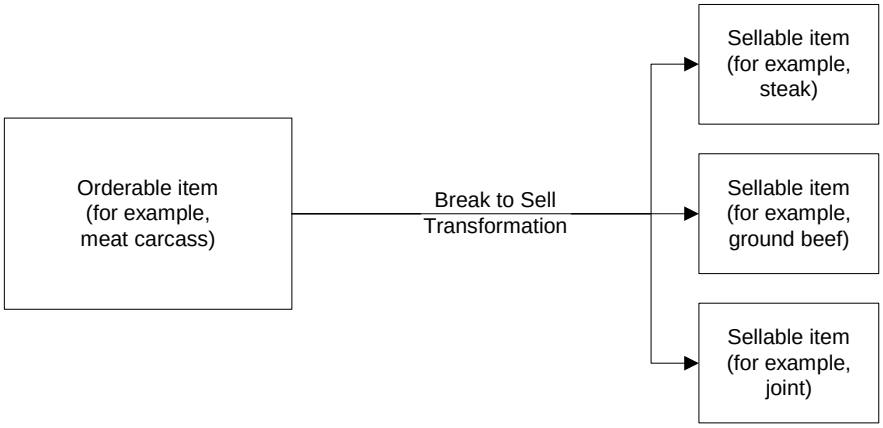
Item transformation allows retailers to manage items where the actual transformation of a product cannot be adequately recorded due to in-store processes.

With product transformation, new 'transform' items are set up as either sellable only or orderable only.

- Sellable only items – A sellable only item has no inventory in the system, so inventory records cannot be viewed from the item maintenance screens. Sellable only items do not hold any supplier links and therefore have no cost prices associated with them.
- Orderable only items – Orderable only items hold inventory, but are not sellable at the POS system. Therefore, no information is sent to the POS system for these items, and no unit retail prices by zone are held for these items.

To hold the relationship between the orderable items and the sellable items, RMS stores the transformation details. These details are used to process sales and inventory transactions for the items.

The following diagram shows how item transformation works:



Item Transformation

Item and Item Component Descriptions

The item message family is a logical grouping for all item data published to the RIB. The components of item messages and their base tables in RMS are:

- Item from the ITEM_MASTER table
- Item-supplier from ITEM_SUPPLIER
- Item-supplier-country from ITEM_SUPP_COUNTRY
- Item-supplier-country-dimension from ITEM_SUPP_COUNTRY_DIM (DIM is the each, inner, pallet, and case dimension for the item, as specified)
- Item-image from ITEM_IMAGE
- Item-UDA identifier-UDA value from UDA_ITEM_LOV (UDA is a user-defined attribute and LOV is list of values)
- Item-UDA identifier from UDA_ITEM_DATE (for the item and UDA date)
- Item-UDA identifier from UDA_ITEM_FF (for UDA, free-format data beyond the values for LOV and date)
- Item-pack components (Bill of Material [BOM]) from PACKITEM_BREAKOUT

- Item UPC reference from ITEM_MASTER.ITEM_NUMBER_TYPE (values held as code type ‘UPCT’ on code_head and code_detail tables)
- Item ticket from ITEM_TICKET

New Item Message Processes

The creation of a new item in RMS begins with an item in a worksheet status on the ITEM_MASTER table. At the time an item is created, other relationships are being defined as well, including the item, supplier, and country relationships, user-defined attributes (UDAs), and others. These item relationship processes in effect become components of a new item message published to the RIB. This section describes the item creation message process and includes the basic item message itself along with the other component relationship messages that become part of the larger item message.

Basic Item Message

As described in the preceding section, item messages can originate in a number of RMS tables. Each of these tables holds a trigger, which fires each time an insert, update, or delete occurs on the table. The new item record itself is displayed on the ITEM_MASTER table. The trigger on this table creates a new message (in this case, a message of the type ItemHdrCre), then calls the message family manager RMSMFM_ITEMS and its ADDTOQ public procedure. ADDTOQ populates the message to the ITEM_MFQUEUE staging table by inserting the following:

- Appropriate value into the message_type column
- Message itself to the message column. Messages are of the data type CLOB (character large object)

New Item Message Publication

The publication of a new item and its components to the RIB is done using a hierarchical message. Here is how the process works:

1. A new item is held on ITEM_MASTER in a status of W (Worksheet) until it is approved.
2. On the ITEM_MFQUEUE staging table, a Worksheet status item is displayed in the message_type column as a value of ItemCre.
3. As the item continues to be built on ITEM_MASTER, an ItemHdrMod value is inserted into the queue’s message_type column.
4. After the item is approved (ITEM_MASTER’s status column value of A [Approved], the trigger causes the insertion of a value of Y (Yes) in the approve_ind column on the queue table.
5. A message with a top-level XML tag of ItemDesc is created that serves as a message wrapper.

At the same time, a sub-message with an XML tag of ItemHdrDesc is also created. This subordinate tag holds a subset of data about the item, most of which is derived from the ITEM_MASTER table.

Subordinate Data and XML Tags

While a new item is being created, item components are also being created. Described earlier in this overview, these component item messages pertain to the item-supplier, item-supplier-country, UDAs, and so on. For example, a new item-supplier record created on ITEM_SUPPLIER causes the trigger on this table to add an ItemSupCre value to the message_type column of the ITEM_MFQUEUE staging table. When the item is approved, a message with an XML tag of ItemSupDesc is added underneath the ItemDesc tag.

Similar processes occur with the other item components. Each component has its own Desc XML tag, for example: ItemSupCtyDesc, ISCDimDesc.

Modify and Delete Messages

Updates and deletions of item data can be included in a larger ItemDesc (item creation) message. If not part of a larger hierarchical message, they are published individually as a flat, non-hierarchical message. Update and delete messages are much smaller than the large hierarchy in a newly created item message (ItemDesc).

Modify Messages

If an existing item record changes on the ITEM_MASTER table, for example, the trigger fires to create an ItemHdrMod message and message type on the queue table. In addition, an ItemHdrDesc message is created. If no ItemCre value already exists in the queue, the ItemHdrDesc message is published to the RIB.

Similarly, item components like item-supplier that are modified result in an ItemSupMod message type inserted on the queue. If an ItemCre and an ItemSupCre already exist, the ItemSupMod is published as part of the larger ItemDesc message. Otherwise, the ItemSupMod is published as an ItemSupDesc message.

Delete Messages

Delete messages are published in the same way that modify messages are. For example, if an item-supplier-country relationship is deleted from RMS' ITEM_SUP_COUNTRY table, the dependent record on ITEM_SUPP_COUNTRY_DIM is also deleted.

1. An ItemSupCtyDel message type is displayed on the item queue table.
2. If the queue already holds an ItemCre or ItemSupCtyCre message, any ItemSupCtyCre and ItemSupCtyMod messages are deleted.

   Otherwise, ItemSupCtyDel is published by itself as an ItemSupCtyRef message to the RIB.

Design Overview

The item message family manager is a package of procedures that adds item family messages to the item queue and publishes these messages for the integration bus to route. Triggers on all the item family tables call a procedure from this package to add a "create", "modify" or "delete" message to the queue. The integration bus calls a procedure in this package to retrieve the next publishable item message from the queue.

All the components that comprise the creation of an item, the item/supplier for example, remain in the queue until the item approval modification message has been published. Any modifications or deletions that occur between item creation in "W" (worksheet) status and "A" (Approved) status are applied to the "create" messages or deleted from the queue as required. For example, if an item UDA is added before item approval and then later deleted before item approval, the item UDA "create" message would be deleted from the queue before publishing the item. If an item/supplier record is updated for a new item before the item is approved, the "create" message for that item/supplier is updated with the new data before the item is published. When the "modify" message that contains the "A" (Approved) status is the next record on the queue, the procedure formats a hierarchical message that contains the item header information and all the child detail records to pass to the integration bus.

Additions, modifications and deletions to item family records for existing approved items are published in the order that they are placed on the queue.

Unless otherwise noted, item publishing includes most of the columns from the item_master table and all of the item family child tables included in the publishing message. Sometimes only certain columns are published, and sometimes additional data is published with the column data from the table row. The item publishing message is built from the following tables:

```
Family Header
item_master  -  transaction level items only
descriptions for the code values
names for department, class and subclass
diff types
base retail price
Item Family Child Tables
item_supplier
item_supp_country
item_supp_country_dim
    descriptions for the code values
item_master  -  reference items
    item, item_number_type, item_parent, primary_ref_ind, format_id, prefix
packitem_breakout
    pack_no, item, packitem_qty
item_image
item_ticket
uda_item_ff
uda_item_lov
uda_item_date
```

Functionality Checklist

Description	RMS	RIB
RMS must publish Item information		
Modify publisher Item	X	X

Form Impact

None.

Business Object Records

Create the following business objects to assist the publishing process:

1. Create a type for a table of rowids.

```
TYPE ROWID_TBL is TABLE OF ROWID;
```

2. Create a record of ROWID_TBL types for keeping track of rowids to update and delete. There should be a ROWID_TBL for ITEM_MFQUEUE deletion, ITEM_MFQUEUE updating, ITEM_PUB_INFO deletion, and ITEMLOC_MFQUEUE deletion.

```
    TYPE ITEM_ROWID_REC is RECORD
(queue_rowid_tbl      ROWID_TBL,
 pub_info_rowid_tbl   ROWID_TBL,
 queue_upd_rowid_tbl  ROWID_TBL,
 itemloc_rowid_tbl    ROWID_TBL
);
```

3. Create a record to assist in publishing the ItemBOM node. This record type was originally in ITEMBOM_XML, but since ITEMBOM_XML is being removed, it is being moved to RMSMFM_ITEMS.

```
TYPE bom_rectype IS RECORD
(pack_no                    VARCHAR2(25),
 seq_no                     NUMBER(4),
 item                       VARCHAR2(25),
 item_parent                VARCHAR2(25),
 pack_tmpl_id               NUMBER(8),
 comp_pack_no               VARCHAR2(25),
 item_qty                   NUMBER(12,4),
 item_parent_pt_qty         NUMBER(12,4),
 comp_pack_qty              NUMBER(12,4),
 pack_item_qty              NUMBER(12,4));

TYPE bom_tabtype is TABLE of bom_rectype
INDEX BY BINARY_INTEGER;
```

Package Impact

Business Object ID

The business object id for item publisher is item, which uniquely identifies an item for publishing.

The RIB uses the business object id to determine message dependencies when sending messages to a subscribing application. If a Create message has already failed in the subscribing application, and a Modify/Delete message is about to be sent from the RIB to the subscribing application, the RIB will not send the modify/delete message if it has the same business object id as the failed Create message. Instead, the Modify/Delete message will go directly to the hospital.

Item type X, item A, message type 'ItemCre' fails in subscriber.

Item type X, item B, message type 'ItemCre' processes successfully in subscriber.

Item type X, item A, message type 'ItemMod' goes directly from RIB to hospital.

Item type X, item B, message type 'ItemMod' goes from RIB to subscriber.

Item type X, item A, message type 'ItemDel' goes directly from RIB to hospital.

Package Name: RMSMFM_ITEMS

Spec File Name: rmsmfm_itemss.pls

Body File Name: rmsmfm_itemsb.pls

Package Specification – Global Variables

```
FAMILY      CONSTANT   RIB_SETTINGS.FAMILY%TYPE      'ITEM';
ITEM_ADD    CONSTANT   VARCHAR2(30)                  'itemcre';
ITEM_UPD    CONSTANT   VARCHAR2(30)                  'itemhdrmod';
ITEM_DEL    CONSTANT   VARCHAR2(30)                  'itemdel';
ISUP_ADD    CONSTANT   VARCHAR2(30)                  'itemsupcre';
ISUP_UPD    CONSTANT   VARCHAR2(30)                  'itemsupmod';
ISUP_DEL    CONSTANT   VARCHAR2(30)                  'itemsupdel';
ISC_ADD     CONSTANT   VARCHAR2(30)                  'itemsupctycre';
ISC_UPD     CONSTANT   VARCHAR2(30)                  'itemsupctymod';
ISC_DEL     CONSTANT   VARCHAR2(30)                  'itemsupctydel';
ISCD_ADD    CONSTANT   VARCHAR2(30)                  'iscdimcre';
ISCD_UPD    CONSTANT   VARCHAR2(30)                  'iscdimmod';
ISCD_DEL    CONSTANT   VARCHAR2(30)                  'iscdimdel';
UPC_ADD     CONSTANT   VARCHAR2(30)                  'itemupccre';
UPC_UPD     CONSTANT   VARCHAR2(30)                  'itemupcmod';
UPC_DEL     CONSTANT   VARCHAR2(30)                  'itemupcdel';
BOM_ADD     CONSTANT   VARCHAR2(30)                  'itembomcre';
BOM_UPD     CONSTANT   VARCHAR2(30)                  'itembommod';
BOM_DEL     CONSTANT   VARCHAR2(30)                  'itembomdel';
UDAF_ADD    CONSTANT   VARCHAR2(30)                  'itemudaffcre';
UDAF_UPD    CONSTANT   VARCHAR2(30)                  'itemudaffmod';
UDAF_DEL    CONSTANT   VARCHAR2(30)                  'itemudaffdel';
UDAD_ADD    CONSTANT   VARCHAR2(30)                  'itemudadatecre';
UDAD_UPD    CONSTANT   VARCHAR2(30)                  'itemudadatemod';
UDAD_DEL    CONSTANT   VARCHAR2(30)                  'itemudadatedel';
UDAL_ADD    CONSTANT   VARCHAR2(30)                  'itemudalovcre';
UDAL_UPD    CONSTANT   VARCHAR2(30)                  'itemudalovmod';
UDAL_DEL    CONSTANT   VARCHAR2(30)                  'itemudalovdel';
IMG_ADD     CONSTANT   VARCHAR2(30)                  'itemimagecre';
IMG_UPD     CONSTANT   VARCHAR2(30)                  'itemimagemod';
```

```
IMG_DEL     CONSTANT    VARCHAR2(30)                'itemimagedel';
TCKT_ADD    CONSTANT    VARCHAR2(30)                'itemtcktcre';
TCKT_DEL    CONSTANT    VARCHAR2(30)                'itemtcktdel';

bom_table  bom_tabtype;
empty_bom  bom_tabtype;
```

Function Level Description – ADDTOQ

```
Function: ADDTOQ
          (O_error_message          OUT     VARCHAR2,
           I_queue_rec              IN      ITEM_MFQUEUE%ROWTYPE,
           I_sellable_ind           IN      ITEM_PUB_INFO.SELLABLE_IND%TYPE,
           I_tran_level_ind         IN      ITEM_PUB_INFO.TRAN_LEVEL_IND%TYPE)
```

This public function puts an item message on ITEM_MFQUEUE for publishing to the RIB. It is called from the item trigger and the detail triggers (ITEM_SUPPLIER, ITEM_SUPP_COUNTRY, ITEM_SUPP_COUNTRY_DIM, PACKITEM, UDA_ITEM, UDA_VALUES, ITEM_IMAGE). The I_queue_rec contains item and, optionally, other detail keys.

For header level insert messages (HDR_ADD), insert a record in the ITEM_PUB_INFO table. The published flag should be set to 'N'. For all message types except header level inserts (HDR_ADD), insert a record into the ITEM_MFQUEUE.

Function Level Description – GETNXT

```
Procedure: GETNXT
              (O_status_code        OUT     VARCHAR2,
               O_error_msg          OUT     VARCHAR2,
               O_message_type       OUT     VARCHAR2,
               O_message            OUT     RIB_OBJECT,
               O_bus_obj_id         OUT     RIB_BUSOBJID_TBL,
               O_routing_info       OUT     RIB_ROUTINGINFO_TBL,
               I_num_threads        IN      NUMBER DEFAULT 1,
               I_thread_val         IN      NUMBER DEFAULT 1)
```

Modify the existing function as follows:

- Change the signature of this package per this specification.
- Replace the code that is in the current function with the functionality in this design.

This public procedure is called from the RIB to get the next messages. It performs a cursor loop on the unpublished records on the ITEM_MFQUEUE table (PUB_STATUS = 'U'). It should only need to execute one loop iteration in most cases. For each record retrieved, GETNXT gets the following:

1. A lock of the queue table for the current business object (item). The lock is obtained by calling the function LOCK_THE_BLOCK. If there are any records on the queue for the current business object that are already locked, the current message is skipped and picked up again in the next loop iteration.
2. A check for records on the queue with a status of 'H'ospital. If there are any such records for the current business object, GETNXT raises an exception to send the current message to the Hospital.
3. Get the published indicator from the ITEM_PUB_INFO table.
4. Call PROCESS_QUEUE_RECORD with the current business object.

The loop will need to execute more than one iteration for the following cases:

1. When a header delete message exists on the queue for a business object that has not been initially published. In this case, simply remove the header delete message from the queue and loop again.
2. The queue is locked for the current business object. This can occur because ADDTOQ, which is called from the triggers, deletes from the queue table for DTL_UPD, DTL_DEL, and HDR_DEL messages.

The information from the ITEM_MFQUEUE and ITEM_PUB_INFO table is passed to PROCESS_QUEUE_RECORD. PROCESS_QUEUE_RECORD will build the Oracle Object message to pass back to the RIB. If PROCESS_QUEUE_RECORD does not run successfully, GETNXT raises an exception.

If any exception is raised in GETNXT, including the exception raised by an unsuccessful call to PROCESS_QUEUE_RECORD, HANDLE_ERRORS is called.

If PROCESS_QUEUE_RECORD fails, the record that keeps track of which mfqueue records to delete/update should be reset. Therefore, a snapshot of the struct is taken before the call to PROCESS_QUEUE_RECORD. If the function fails, the record is reset back to the snapshot.

Function Level Description – PUB_RETRY

```
Procedure: PUB_RETRY
                (O_status_code       OUT                  VARCHAR2,
                O_error_msg          OUT                  VARCHAR2,
                O_message            OUT                  RIB_OBJECT,
                O_message_type       IN  OUT              VARCHAR2,
                O_bus_obj_id         IN  OUT NOCOPY       RIB_BUSOBJID_TBL,
                O_routing_info       IN  OUT NOCOPY       RIB_ROUTINGINFO_TBL)
```

This public procedure performs the same tasks as GETNXT except that it only loops for a specific row in the ITEM_MFQUEUE table. The record on ITEM_MFQUEUE must match the passed in sequence number (contained in the ROUTING_INFO).

Function Level Description – PROCESS_QUEUE_RECORD (local)

This private function controls the building of Oracle Objects (DESC or REF) given the business transaction's key values and a message type. It contains all of the shared processing between GETNXT and PUB_RETRY.

Get relevant publishing info for the item in ITEM_PUB_INFO, including the published indicator and approved upon create indicator.

If I_hdr_published is either 'N' (not published)

- If I_hdr_published is 'N', check to see if the current message should cause the item to be published. This will be true if the status has changed to 'A'pproved or if an ITEM_SUPP_COUNTRY record has been added to an item that was approved upon create. If the item is ready to be published for the first time, the message type is a header create (HDR_ADD). If it is not ready to be published, add the record's ROWID to the structure that keeps track of ROWIDs to delete.
- Call MAKE_CREATE to build the DESC Oracle Object to publish to the RIB. This will also take care of any ITEM_MFQUEUE deletes, updating ITEM_PUB_INFO.PUBLISHED to 'Y' or 'I', and bulk updating the detail tables publish_ind column to 'Y' for those detail rows that have been published.

If the message type is an update or create message type at any level (for example, ITEM_ADD, ISUP_ADD, ISUP_UPD, and so on)

- Call RMSMFM_ITEMS_BUILD.BUILD_MESSAGE to build the DESC Oracle Object to publish to the RIB.
- RMSMFM_ITEMS_BUILD.BUILD_MESSAGE will return an indicator specifying if the record exists. The record in question is the record on the functional table corresponding to the current MFQUEUE record being processed. For example, for ITEM_ADD or ITEM_UPD message, the record exists indicator specifies whether or not the ITEM_MASTER record for the item still exists. For an ISUP_ADD or ISUP_UPD message, the record exists indicator specifies whether or not the ITEM_SUPPLIER record for the item/supplier combination still exists. If the record does not exist, the current message cannot be published.
  - If the record does not exist and the message type is an update, delete the current MFQUEUE record (that is, add the ROWID to the list of ROWIDs to be eventually deleted.)
  - If the record does not exist and the message type is a create, update the current MFQUEUE record's pub_status to 'N' so that the record will be skipped but remain on the queue (that is, add the ROWID to the list of ROWIDs to be eventually updated.)

If the message type is a delete message type at any level (for example, ITEM_DEL, ISUP_DEL, and so on)

- Call RMSMFM_ITEMS_BUILD.BUILD_DELETE_MESSAGE to build the REF Oracle Object to publish to the RIB.
- For the current delete message, there could be a corresponding create message earlier on the queue if the create message could not be published (see update/create message type section above.) If there is a corresponding create message earlier on the queue, delete both create and delete messages (that is, add the ROWIDs to the list of ROWIDs to be eventually deleted), and do not publish anything.

Finally, perform DML cleanup if a message is going to be published.

- Call UPDATE_QUEUE_TABLE to perform DML using the global record that keeps track of QUEUE records to update/delete.
- If the message type is ITEM_ADD, update the item's ITEM_PUB_INFO to published = 'Y'.
- If the message type is ITEM_DEL, delete the item's ITEM_PUB_INFO record.

Function Level Description – MAKE_CREATE (local)

This private function is used to create the Oracle Object for the initial publication of a business transaction. I_business_object contains the item header key values (item). I_rowid is the rowid of the item_mfqueue row fetched from GETNXT.

- Call BUILD_HEADER_OBJECT to get a header level Oracle Object.
- Call BUILD_DETAIL_OBJECTS to get a table of detail level Oracle objects and a table of ITEM_MFQUEUE rowids to delete with and a table of detail table rowids to update publish_ind with.
- Update ITEM_PUB_INFO.published to 'Y' or 'I' depending on if all details are published.

- Delete records from the ITEM_MFQUEUE for all rowids returned by BUILD_DETAIL_OBJECTS. Deletes are done by rowids instead of business transaction keys to ensure that nothing is deleted off the queue that has not been published.
- If the entire business transaction was added to the Oracle Object, also delete the ITEM_MFQUEUE record that was picked up by GETNXT. If the entire business transaction was **not** published, the system must leave something on the ITEM_MFQUEUE to ensure that the rest of it is picked up by the next call to GETNXT.
- Update the detail tables publish_ind column to 'Y' by each detail table of rowids returned from BUILD_DETAIL_OBJECTS.
- The header and detail level Oracle Objects are combined and returned.

Function Level Description – HANDLE_ERRORS (local)

This private procedure is called from GETNXT and PUB_RETRY when an exception is raised. I_seq_no is the sequence number of the driving ITEM_MFQUEUE record. I_function_keys contains detail level key values (item and optional detail keys).

If the error is a non-fatal error, HANDLE_ERRORS passes the sequence number of the driving ITEM_MFQUEUE record back to the RIB in the ROUTING_INFO. It sends back a status of 'H'ospital to the RIB as well. It then updates the status of the queue record to 'H'ospital, so that it will not get picked up again by the driving cursor in GETNXT.

If the error is a fatal error, a status of 'E'rror is returned to the RIB. The error is considered non-fatal if no DML has occurred yet. Whenever DML has occurred, then the global variable LP_error_status is flipped from 'H'ospital to 'E'rror.

Package Name: RMSMFM_ITEMS_BUILD

Spec File Name: rmsmfm_items.pls

Body File Name: rmsmfm_itemb.pls

Function Level Description – BUILD_MESSAGE

```
Function: BUILD_MESSAGE
    (O_error_msg          OUT               VARCHAR2,
     O_message            IN OUT NOCOPY     RIB_ITEMDESC_REC,
     O_rowids_rec         IN OUT NOCOPY     ROWIDS_REC,
     O_record_exists      IN OUT            BOOLEAN,
     I_message_type       IN                ITEM_MFQUEUE.MESSAGE_TYPE%TYPE,
     I_tran_level_ind     IN                ITEM_PUB_INFO.TRAN_LEVEL_IND%TYPE,
     I_queue_rec          IN                ITEM_MFQUEUE%ROWTYPE)
```

The private function is responsible for building detail level DESC Oracle Objects. It builds as many detail Oracle Object as it can given the passed in message type and business object keys (item).

Call the following:

- BUILD_HEADER_DETAIL
- BUILD_SUPPLIER_DETAIL
- BUILD_COUNTRY_DETAIL
- BUILD_DIM_DETAIL
- BUILD_UDA_LOV_DETAIL

- BUILD_UDA_FF_DETAIL
- BUILD_UDA_DATE_DETAIL
- BUILD_IMAGE_DETAIL
- BUIILD_UPC_DETAIL
- BUILD_BOM_DETAIL
- BUILD_TICKET_DETAIL

Function Level Description – BUILD_DELETE_MESSAGE

```
Function: BUILD_DETAIL_CHANGE_OBJECTS
           (O_error_msg       OUT            VARCHAR2,
            O_message         IN OUT NOCOPY  RIB_ITEMDESC_REC,
            I_message_type    IN             ITEM_MFQUEUE.MESSAGE_TYPE%TYPE,
            I_business_obj    IN             ITEM_KEY_REC)
```

This function builds a REF Oracle Object to publish to the RIB for all delete message types (for example, ITEM_DEL, ISUP_DEL, ISC_DEL, and so on).

The function also checks to see if there is a corresponding Create message for the current delete message. If so, O_create_rowid is set. This is used to determine if the Delete message should be published (see PROCESS_QUEUE_RECORD description above). If both Create and Delete messages are on the queue, neither is published.

Detail create and detail update messages (DTL_ADD, DTL_UPD). I_business_obj contains the header level key values (item).

Function Level Description – BUILD_HEADER_OBJECT (local)

This private function accepts item header key values (item), builds and returns a header level DESC Oracle Object. Call GET_ITEM_INFO to retrieve data supplementary to ITEM_MASTER. If the item is not found on ITEM_MASTER, O_record_exists is set to FALSE.

Function Level Description – BUILD DETAIL functions (all local)

The following functions have the same format:

- BUILD_SUPPLIER_DETAIL
- BUILD_COUNTRY_DETAIL
- BUILD_DIM_DETAIL
- BUILD_UDA_LOV_DETAIL
- BUILD_UDA_FF_DETAIL
- BUILD_UDA_DATE_DETAIL
- BUILD_IMAGE_DETAIL
- BUIILD_UPC_DETAIL
- BUILD_BOM_DETAIL
- BUILD_TICKET_DETAIL

They have the same specifications, except as noted below.

The functions for building detail nodes for the ITEMDESC message work in the same way. The functions build as many detail Oracle Objects as they can given the passed in message type and business object keys.

The difference between the different detail functions lies in the data being accessed. BUILD_SUPPLIER_DETAIL retrieves information from ITEM_SUPPLIER,

BUILD_COUNTRY_DETAIL retrieves information from ITEM_SUPP_COUNTRY, etc.

BUILD_SUPPLlER_DETAIL and BUILD_COUNTRY_DETAIL are the only functions that have the input parameter I_orderable_item. This is used to validate orderable items. If an item is orderable, and the initial ITEM_ADD message is being created, at least one supplier node and one supplier/country node are required. **This is the only business validation done by the item publisher.**

If the original create message is being published (I_message_type would be ITEM_ADD)

- Select all detail records for the business transaction. Return a table of ITEM_MFQUEUE rowids for each message that is placed into the Oracle Object.
- Since the message being published is ITEM_ADD, there may not be a record on the MFQUEUE table for each detail record that needs to be retrieved. Therefore, no inner join to the MFQUEUE table is done. However, if there are any MFQUEUE records for details, they should be deleted. Therefore, a UNION to a second query is done to select all relevant MFQUEUE records for deletion.

If the message being published is a detail add or detail update (for example, ISUP_ADD, ISUP_UPD, ISC_ADD, ISC_UPD)

- Select all detail records for the business transaction. Return a table of ITEM_MFQUEUE rowids for each message that is placed into the Oracle Object.
- Since the message being published is a detail create or update, the only details that should be added to the message are those details that have a record on the MFQUEUE table. Therefore, an inner join between the MFQUEUE table and the business detail table is performed. Any MFQUEUE records retrieved will have their ROWIDs added to the list of ROWIDs that will eventually be deleted.
- If no records are retrieved for the detail record query, O_records_exist is set to FALSE.

A concern here is making sure that the system does not delete information from the queue table that has not been published. For this reason, the system does deletes by ROWID. The system also tries to get everything in the same cursor to ensure that the message published matches the deletes that are performed from the ITEM_MFQUEUE table regardless of trigger execution during GETNXT calls.

Function Level Description – GET_ITEM_INFO (local)

This private function gets ITEM_MASTER as input and retrieves supplementary data. For example, each item has a department, class, and subclass. GET_ITEM_INFO will retrieve the descriptions for these three fields. This function is called from BUILD_HEADER_OBJECT.

Function Level Description – BUILD_DIMENSION_DESCRIPTIONS (local)

This private function is similar to GET_ITEM_INFO in that it retrieves supplementary data. This function, however, is called when item/supplier/country/dimension message nodes are being populated. This function is called from BUILD_DIM_DETAIL.

Trigger Impact

Trigger Name: EC_TABLE_IEM_AIUDR.TRG (mod)

Trigger File Name: ec_table_iem_aiudr.trg (mod)

Table: ITEM_MASTER

Modify the trigger on the ITEM table to capture Inserts, Updates, and Deletes. Remove all of the code except the code that checks the item_level and tran_level. This is needed to determine which message type to send to the queue, item or UPC (reference item).

Inserts

- Send the header level item info to the ADDTOQ procedure in the MFM with the message type RMSMFM_ITEM.ITEM_ADD or RMSMFM_ITEM.UPC_ADD.

Updates

- Send the header level item info to the ADDTOQ procedure in the MFM with the message type RMSMFM_ITEM.ITEM_UPD or RMSMFM_ITEM.UPC_UPD.

Deletes

- Send the header level info to the ADDTOQ procedure in the MFM with the message type RMSMFM_ITEM.ITEM_DEL or RMSMFM_ITEM.UPC_DEL.

In all these cases, build the function keys for ADDTOQ with item.

Trigger Name: EC_TABLE_ISP_AIUDR.TRG (mod)

Trigger File Name: ec_table_isp_aiudr.trg (mod)

Table: ITEM_SUPPLIER

Populate the ITEM_MFQUEUE table according to the message type. Make sure that only transaction level items are added to the ITEM_MFQUEUE table.

Inserts

- Send the detail level info to the ADDTOQ procedure in the MFM with the message type RMSMFM_ITEM.DTL_ADD

Updates

- Send the detail level info to the ADDTOQ procedure in the MFM with the message type RMSMFM_ITEM.DTL_UPD.

Deletes

- Send the detail level info to the ADDTOQ procedure in the MFM with the message type RMSMFM_ITEM.DTL_DEL.

In all these cases, build the function keys for ADDTOQ with item and supplier.

Trigger Name: EC_TABLE_ISC_AIUDR.TRG (mod)

Trigger File Name: ec_table_isc_aiudr.trg (mod)

Table: ITEM_SUPP_COUNTRY

Populate the ITEM_MFQUEUE table according to the message type. Make sure that only transaction level items are added to the ITEM_MFQUEUE table.

Inserts

- Send the detail level info to the ADDTOQ procedure in the MFM with the message type RMSMFM_ITEM.DTL_ADD.

Updates

- Send the detail level info to the ADDTOQ procedure in the MFM with the message type RMSMFM_ITEM.DTL_UPD.

Deletes

- Send the detail level info to the ADDTOQ procedure in the MFM with the message type RMSMFM_ITEM.DTL_DEL.

In all these cases, build the function keys for ADDTOQ with item, supplier and origin_country_id.

Trigger Name: EC_TABLE_ISD_AIUDR.TRG (mod)

Trigger File Name: ec_table_isd_aiudr.trg (mod)

Table: ITEM_SUPP_COUNTRY_DIM

Populate the ITEM_MFQUEUE table according to the message type. Make sure that only transaction level items are added to the ITEM_MFQUEUE table.

Inserts

- Send the detail level info to the ADDTOQ procedure in the MFM with the message type RMSMFM_ITEM.DTL_ADD

Updates

- Send the detail level info to the ADDTOQ procedure in the MFM with the message type RMSMFM_ITEM.DTL_UPD.

Deletes

- Send the detail level info to the ADDTOQ procedure in the MFM with the message type RMSMFM_ITEM.DTL_DEL.

In all these cases, build the function keys for ADDTOQ with item, supplier, origin_country_id.

Trigger Name: EC_TABLE_PKS_AIUDR.TRG (mod)

Trigger File Name: ec_table_pks_aiudr.trg (mod)

Table: PACKITEM_BREAKOUT

This trigger captures inserts, updates and deletes on the table. It populates a PL/SQL table of records, RMSMFM_ITEMS.BOM_TABLE, which will be used in the statement trigger to build an XML message and place it on the item queue.

Trigger Name: EC_TABLE_PKS_IUDS.TRG (mod)

Trigger File Name: ec_table_pks_aiudr.trg (mod)

Table: PACKITEM_BREAKOUT

This trigger will group all of the data currently stored in the PL/SQL table of records populated by the EC_TABLE_PKS_AIUDR trigger, and call RMSMFM_ADDTOQ for every pack component in the table of records.

Trigger Name: EC_TABLE_UIT_AIUDR.TRG (mod)

Trigger File Name: ec_table_uit_aiudr.trg (mod)

Table: UDA_ITEM_DATE

Populate the ITEM_MFQUEUE table according to the message type. Make sure that only transaction level items are added to the ITEM_MFQUEUE table.

Inserts

- Send the detail level info to the ADDTOQ procedure in the MFM with the message type RMSMFM_ITEM.DTL_ADD.

Updates

- Send the detail level info to the ADDTOQ procedure in the MFM with the message type RMSMFM_ITEM.DTL_UPD.

Deletes

- Send the detail level info to the ADDTOQ procedure in the MFM with the message type RMSMFM_ITEM.DTL_DEL.

In all these cases, build the function keys for ADDTOQ with item, uda_id.

Trigger Name: EC_TABLE_UIF_AIUDR.TRG (mod)

Trigger File Name: ec_table_uif_aiudr.trg (mod)

Table: UDA_ITEM_FF

Populate the ITEM_MFQUEUE table according to the message type. Make sure that only transaction level items are added to the ITEM_MFQUEUE table.

Inserts

- Send the detail level info to the ADDTOQ procedure in the MFM with the message type RMSMFM_ITEM.DTL_ADD.

Updates

- Send the detail level info to the ADDTOQ procedure in the MFM with the message type RMSMFM_ITEM.DTL_UPD.

Deletes

- Send the detail level info to the ADDTOQ procedure in the MFM with the message type RMSMFM_ITEM.DTL_DEL.

In all these cases, build the function keys for ADDTOQ with item, uda_id.

Trigger Name: EC_TABLE_UIL_AIUDR.TRG (mod)

Trigger File Name: ec_table_uil_aiudr.trg (mod)

Table: UDA_ITEM_LOV

Populate the ITEM_MFQUEUE table according to the message type. Make sure that only transaction level items are added to the ITEM_MFQUEUE table.

Inserts

- Send the detail level info to the ADDTOQ procedure in the MFM with the message type RMSMFM_ITEM.DTL_ADD

Updates

- Send the detail level info to the ADDTOQ procedure in the MFM with the message type RMSMFM_ITEM.DTL_UPD.

Deletes

- Send the detail level info to the ADDTOQ procedure in the MFM with the message type RMSMFM_ITEM.DTL_DEL.

In all these cases, build the function keys for ADDTOQ with item, uda_id and uda_value.

DTD

Here are the filenames that correspond with each message type. Please consult RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
itemcre	Item Create Message	ItemDesc.dtd
itemmod	Item Modify Message	ItemDesc.dtd
itemdel	Item Delete Message	ItemRef.dtd
itemsupcre	Item Supplier Create Message	ItemSupDesc.dtd
itemsupmod	Item Supplier Modify Message	ItemSupDesc.dtd
itemsupdel	Item Supplier Delete Message	ItemSupRef.dtd
itemsupctycre	Item Supplier Country Create Message	ItemSupCtyDesc.dtd
itemsupctymod	Item Supplier Country Modify Message	ItemSupCtyDesc.dtd
itemsupctydel	Item Supplier Country Delete Message	ItemSupCtyRef.dtd
iscdimcre	Item Supplier Country Dimension Create Message	ISCDimDesc.dtd
iscdimmod	Item Supplier Country Dimension Modify Message	ISCDimDesc.dtd

Message Types	Message Type Description	Document Type Definition (DTD)
iscdimdel	Item Supplier Country Dimension Delete Message	ISCDimRef.dtd
itemupccre	Item UPC Create Message	ItemUPCDesc.dtd
itemupcmod	Item UPC Modify Message	ItemUPCDesc.dtd
itemupcdel	Item UPC Delete Message	ItemUPCRef.dtd
itembomcre	Item BOM Create Message	ItemBOMDesc.dtd
itembommod	Item BOM Modify Message	ItemBOMDesc.dtd
itembomdel	Item BOM Delete Message	ItemBOMRef.dtd
itemudaffcre	Item UDA Free Form TextCreate Message	ItemUDAFFDesc.dtd
itemudaffmod	Item UDA Free Form Text Modify Message	ItemUDAFFDesc.dtd
itemudaffdel	Item UDA Free Form Text Delete Message	ItemUDAFFRef.dtd
itemudalovcre	Item UDA LOV Create Message	ItemUDALOVDesc.dtd
itemudalovmod	Item UDA LOV Modify Message	ItemUDALOVDesc.dtd
itemudalovdel	Item UDA LOV Delete Message	ItemUDALOVRef.dtd
itemudadatecre	Item UDA Date Create Message	ItemUDADateDesc.dtd
itemudadatemod	Item UDA Date Modify Message	ItemUDADateDesc.dtd
itemudadatedel	Item UDA Date Delete Message	ItemUDADateRef.dtd
itemimagecre	Item Image Create Message	ItemImageDesc.dtd
itemimagemod	Item Image Modify Message	ItemImageDesc.dtd
itemimagedel	Item Image Delete Message	ItemImageRef.dtd

Table Impact

TABLE	SELECT	INSERT	UPDATE	DELETE
ITEM_MFQUEUE	Yes	Yes	Yes	Yes
ITEM_PUB_INFO	Yes	Yes	Yes	Yes
ITEMLOC_MFQUEUE	Yes	No	No	Yes
ITEM_MASTER	Yes	No	No	No
ITEM_SUPPLIER	Yes	No	No	No
ITEM_SUPP_COUNTRY	Yes	No	No	No
ITEM_SUPP_COUNTRY_DIM	Yes	No	No	No

TABLE	SELECT	INSERT	UPDATE	DELETE
UDA_ITEM_LOV	Yes	No	No	No
UDA_ITEM_DATE	Yes	No	No	No
UDA_ITEM_FF	Yes	No	No	No
ITEM_IMAGE	Yes	No	No	No
PACKITEM_BREAKOUT	Yes	No	No	No
V_PACKSKU_QTY	Yes	No	No	No
ITEM_TICKET	Yes	No	No	No
CODE_DETAIL	Yes	No	No	No
DEPS	Yes	No	No	No
CLASS	Yes	No	No	No
SUBCLASS	Yes	No	No	No
V_DIFF_ID_GROUP_TYPE	Yes	No	No	No
ITEM_ZONE_PRICE	Yes	No	No	No
PACKITEM	Yes	No	No	No

Design Assumptions

- It is not possible for a detail trigger to accurately know the status of a header table.
- In order for the detail triggers to accurately know when to add a message to the queue, RMS should not allow approval of a business object while detail modifications are being made.
- It is not possible for a header trigger or a detail trigger to know the status of anything modified by GETNXT. If a header trigger or detail trigger is trying to delete queue records that GETNXT currently has locked, it will have to wait until GETNXT is finished and removes the lock. It is assumed that this time will be fairly short (at most 2-3 seconds.) It is also assumed that this will occur rarely, as it involves updating/deleting detail records on a business object that has already been approved. This also has to occur at the same time GETNXT is processing the current business object.
- Push off all DML statements as late as possible. Once DML statements have taken place, any error becomes a fatal error rather than a hospital error.

Item Location

Business Overview

RMS defines and publishes item-location relationships. The published message includes the retail fields as a starting set of retails for each item-location. Also published is a store price indicator for the item-location relationship. This indicator specifies whether or not that store (location) can mark down the price of the item. Subsequent price changes need to be taken from the RPM application.

RMS also provides the initial retail for external subsystems. While all subsequent price changes will be taken from RPM, external subsystems need RMS to send a starting set of retails for each item-location. To meet this requirement, retail fields are part of the item-location message. These fields will be published upon creation; subsequent updates to these retail fields, however, will not trigger an update message.

The item-location publisher also publishes warehouses as well as stores.

Functionality Checklist

Description	RMS	RIB
RMS must publish item loc information		
Create new Publisher	X	X

Form Impact

None.

Business Object Records

None.

Package Impact

- Item Loc publishing – store price initialized the publishing of store_price on Item_LOC.
- Item Loc publishing – store price ind to verify the initial publishing of pricing info (unit retail, selling unit retail and uom) on ITEM_LOC.
- Item Loc publishing – pricing, make an update of the pricing info on ITEM_LOC – should NOT be published.

Package Name: RMSMFM_ITEMLOC

Spec File Name: rmsmfm_itemlocs.pls

Body File Name: rmsmfm_itemlocb.pls

Package Specification – Global Variables

```
FAMILY            CONSTANT  VARCHAR2(64)          'ItemLoc';
ITEMLOC_ADD       CONSTANT  VARCHAR2(20)          'ItemLocCre';
ITEMLOC_UPD       CONSTANT  VARCHAR2(20)          'ItemLocMod';
ITEMLOC_DEL       CONSTANT  VARCHAR2(20)          'ItemLocDel';
REPL_UPD          CONSTANT  VARCHAR2(20)          'ItemLocReplMod';
```

Function Level Description – ADDTOQ

```
Function:   ADDTOQ
   (O_error_message          OUT    VARCHAR2,
    I_message_type           IN     ITEMLOC_MFQUEUE.MESSAGE_TYPE%TYPE,
    I_itemloc_record         IN     ITEM_LOC%ROWTYPE,
    I_prim_repl_supplier     IN     REPL_ITEM_LOC.PRIMARY_REPL_SUPPLIER%TYPE,
    I_repl_method            IN     REPL_ITEM_LOC.REPL_METHOD%TYPE,
    _reject_store_ord_ind    IN     REPL_ITEM_LOC.REJECT_STORE_ORD_IND%TYPE,
    I_next_delivery_date     IN     REPL_ITEM_LOC.NEXT_DELIVERY_DATE%TYPE);
```

This will call the API_LIBRARY.GET_RIB_SETTINGS if the LP_num_threads is NULL and insert the family record into ITEMLOC_MFQUEUE table. The call for HASH_ITEM will insert the I_itemloc_record.item information into ITEMLOC_MFQUEUE table.

Function Level Description – GETNXT

```
Procedure: GETNXT ( O_status_code     OUT          VARCHAR2,
           O_error_msg                OUT          VARCHAR2,
           O_message_type             OUT          VARCHAR2,
           O_message                  OUT          RIB_OBJECT,
           O_bus_obj_id               OUT          RIB_BUSOBJID_TBL,
           O_routing_info             OUT          RIB_ROUTINGINFO_TBL,
           I_num_threads              IN           NUMBER DEFAULT 1,
           I_thread_val               IN           NUMBER DEFAULT 1);
```

Make sure to initialize LP_error_status to API_CODES.HOSPITAL at the beginning of GETNXT.

The RIB calls GETNXT to get messages. The driving cursor will query for unpublished records on the ITEMLOC_MFQUEUE table (PUB_STATUS = ‘U’).

Because ITEMLOC records should not be published before ITEM records, include a clause in the driving cursor that checks for ITEM CREATE messages on the ITEM_MFQUEUE table. The ITEMLOC_MFQUEUE record should not be selected from the driving cursor if the ITEM CREATE message still exists on ITEM_MFQUEUE. Also, ITEMLOC_MFQUEUE cleanup should be included in ITEM_MFQUEUE cleanup. When the item publisher RMSMFM_ITEMS encounters a DELETE message for an item that has never been published, it deletes all records for the item from the ITEM_MFQUEUE table. This is done in the program unit CLEAN_QUEUE. CLEAN_QUEUE should now also delete from ITEMLOC_MFQUEUE when a DELETE message for a non-published item is encountered.

After retrieving a record from the queue table, GETNXT should check for records on the queue with a status of ‘H’ospital. If there are any such records for the current business object, GETNXT should raise an exception to send the current message to the hospital.

The information from the ITEMLOC_MFQUEUE table is passed to PROCESS_QUEUE_RECORD. PROCESS_QUEUE_RECORD will build the Oracle Object message to pass back to the RIB. If PROCESS_QUEUE_RECORD does not run successfully, GETNXT should raise an exception.

If any exception is raised in GETNXT, including the exception raised by an unsuccessful call to PROCESS_QUEUE_RECORD, HANDLE_ERRORS should be called.

Function Level Description – PUB_RETRY

```
Procedure: PUB_RETRY
              (O_status_code        OUT           VARCHAR2,
               O_error_msg          OUT           VARCHAR2,
               O_message            OUT           RIB_OBJECT,
               O_message_type       IN  OUT       VARCHAR2,
               O_bus_obj_id         IN  OUT       RIB_BUSOBJID_TBL,
               O_routing_info       IN  OUT       RIB_ROUTINGINFO_TBL,
               I_REF_OBJECT         IN            RIB_OBJECT);
```

Same as GETNXT except:

The record on ITEMLOC_MFQUEUE must match the passed in sequence number (contained in the ROUTING_INFO).

Function Level Description – PROCESS_QUEUE_RECORD (local)

This function controls the building of Oracle Objects given the business transaction's key values and a message type. It contains all of the shared processing between GETNXT and PUB_RETRY.

If the record from ITEMLOC_MFQUEUE table is an add or update (ITEMLOC_ADD, ITEMLOC_UPD)

- Call BUILD_DETAIL_OBJECTS to build the Oracle Object to publish to the RIB. This will also take care of any ITEMLOC_MFQUEUE deletes and ROUTING_INFO logic.

If the record from ITEMLOC_MFQUEUE table is a delete (ITEMLOC_DEL)

- Call BUILD_DETAIL_DELETE_OBJECTS to build the Oracle Object to publish to the RIB. This will also take care of any ITEMLOC_MFQUEUE deletes and the ROUTING_INFO logic.

Function Level Description – BUILD_DETAIL_OBJECTS (local)

The function is responsible for the Oracle Object used for a DESC message (inserts and updates.) It adds as many mfqueue records to the message as it can; given the passed-in message type and business object keys.

- Select all records on the ITEMLOC_MFQUEUE that are for the same item. Fetch the records in order of seq_no on the MFQUEUE table. Fetch the records into a table using BULK COLLECT, with MAX_DETAILS_TO_PUBLISH as the LIMIT clause.
- Loop through records in the BULK COLLECT table. If the record's message_type differs from the message type passed into the function, exit from the loop. Otherwise, add the data from the record to the Oracle Object being used for publication.
- Ensure that ITEMLOC_MFQUEUE is deleted from as needed.
- Ensure that ROUTING_INFO is constructed if routing information is stored at the detail level in the business transaction.

Make sure to set LP_error_status to API_CODES.UNHANDLED_ERROR before any DML statements.

A concern here is making sure that the system does not delete records from the queue table that have not been published. For this reason, the system performs deletes by ROWID. The system also tries to get everything in the same cursor. This should ensure that the message published matches the deletes performed from the ITEMLOC_MFQUEUE table regardless of trigger execution during GETNXT calls.

Function Level Description – BUILD_DETAIL_DELETE_OBJECTS (local)

This function works the same way as BUILD_DETAIL_OBJECTS, except for the fact that a REF object is being created instead of a DESC object.

Function Level Description – HANDLE_ERRORS (local)

HANDLE_ERRORS is called from GETNXT and PUB_RETRY when an exception is raised.

If the error is a non-fatal error, GETNXT passes the sequence number of the driving ITEMLOC_MFQUEUE record back to the RIB in the ROUTING_INFO. It sends back a status of ‘H’ospital to the RIB as well. It then updates the status of the queue record to ‘H’ospital, so that it will not get picked up again by the driving cursor in GETNXT.

If the error is a fatal error, a status of ‘E’rror is returned to the RIB.

The error is considered non-fatal if no DML has occurred yet. Whenever DML has occurred, then the global variable LP_error_status is flipped from ‘H’ospital to ‘E’rror.

Trigger Impact

Create a trigger on the ITEM_LOC to capture inserts, updates, and deletes.

Only transaction-level items should be processed. If the item is not transaction-level, exit the trigger before calling ADDTOQ

Trigger Name: EC_TABLE_ITL_AIUDR.TRG (mod)

Trigger File Name: ec_table_itl_aiudr.trg (mod)

Table: ITEMLOC

Inserts

- Send the L_record (I_item, I_loc, and the I_loc_type) to the ADDTOQ procedure in the MFM with the message type RMSMFM_ITEMLOC.ITEMLOC_ADD.

Updates

- Send the L_prim_repl_supplier , L_repl_method, L_reject_store_ord_ind,L_next_delivery_date to the ADDTOQ procedure in the MFM with the message type RMSMFM_ITEMLOC.ITEMLOC_UPD.
- The only updates that need to be captured are updates to the columns receive_as_type, source_wh, store_price_ind, primary_supp, status, source_method, local_item_desc, primary_cntry, local_short_desc, and taxable_ind.

Deletes

- Send the L_record (I_item, I_loc, and the I_loc_type) to the ADDTOQ procedure in the MFM with the message type RMSMFM_ITEMLOC.ITEMLOC_DEL.

The trigger should fire not only for stores (loc_type = 'S') but also for warehouses (loc_type = 'W').

Trigger Name: EC_TABLE_RIL_AIUDR.TRG (mod)

Trigger File Name: ec_table_ril_aiudr.trg (mod)

Table: REPL_ITEM_LOC

Create a trigger on the ITEM_LOC to capture inserts, updates, and deletes.

Updates

- Send the L_prim_repl_supplier , L_repl_method, L_reject_store_ord_ind,L_next_delivery_date and the L_record (I_item, I_loc, and the I_loc_type) to the ADDTOQ procedure in the MFM with the message type RMSMFM_ITEMLOC.REPL_UPD.
- The only updates that need to be captured are updates to the columns primary_repl_supplier, repl_method, reject_store_ord_ind, and next_delivery_date.

Deletes

- Send the L_record (I_item, I_loc, and the I_loc_type) to the ADDTOQ procedure in the MFM with the message type RMSMFM_ITEMLOC.REPL_UPD

DTD

Here are the filenames that correspond with each message type. Please consult the RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
ItemLocCre	Item Loc Create Message	ItemLocDesc.dtd
ItemLocMod	Item Loc Modify Message	ItemLocDesc.dtd
ItemLocDel	Item Loc Delete Message	ItemLocRef.dtd
ItemLocReplMod	Item Loc Replenishment Modify Message	ItemLocDesc.dtd

Table Impact

TABLE	SELECT	INSERT	UPDATE	DELETE
ITEM_MFQUEUE	Yes	No	No	No
ITEMLOC_MFQUEUE	Yes	Yes	Yes	Yes
ITEM_MASTER	Yes	No	No	No

Design Assumptions

- It is not possible for a detail trigger to accurately know the status of a header table.
- In order for the detail triggers to accurately know when to add a message to the queue, RMS should not allow approval of a business object while detail modifications are being made.
- It is not possible for a header trigger or a detail trigger to know the status of anything modified by GETNXT. If a header trigger or detail trigger is trying to delete queue records that GETNXT currently has locked, it will have to wait until GETNXT is finished and removes the lock. It is assumed that this time will be fairly short (at most 2-3 seconds.) It is also assumed that this will occur rarely because it involves updating/deleting detail records on a business object that has already been approved. This also has to occur at the same time GETNXT is processing the current business object.
- Push off all DML statements as late as possible. Once DML statements have taken place, any error becomes a fatal error rather than a hospital error.

Merchandise Hierarchy

Business Overview

RPM must know the merchandise hierarchy values that RMS contains. To ensure that RPM has the most current merchandise hierarchy values that RMS has, a publishing API sends the merchandise hierarchy information to the RIB so that RPM may subscribe to it.

Functionality Checklist

Description	RMS	RIB
RMS must publish Merchandise Hierarchy information		
Create new Publisher	X	X

Form Impact

None.

Business Object Records

N/A

Package Impact

Business Object ID

The RIB uses the business object ID to determine message dependencies when sending messages to a subscribing application. If a create message has already failed in the subscribing application, and a modify/delete message is about to be sent from the RIB to the subscribing application, the RIB will not send the modify/delete message if it has the same business object ID as the failed create message. Instead, the modify/delete message will go directly to the hospital.

If the message relates to districts, the business object ID will be the district. If the message relates to groups, the business object ID will be the group number. If the message relates to a department, the department number is the business object ID. If the message relates to a class, the business object ID will be the department number and the class number. Finally, if the message relates to a subclass, the business object ID will be the department, class and subclass.

Package Name: RMSMFM_MERCHHIER

Spec File Name: rmsmfm_merchhiers.pls

Body File Name: rmsmfm_merchhierb.pls

Package Specification – Global Variables

```
FAMILY     CONSTANT   RIB_SETTINGS.FAMILY%TYPE      := 'merchhier';

DIV_ADD    CONSTANT   VARCHAR2(64)          := 'divisioncre';
DIV_UPD    CONSTANT   VARCHAR2(64)          := 'divisionmod';
DIV_DEL    CONSTANT   VARCHAR2(64)          := 'divisiondel';

GRP_ADD    CONSTANT   VARCHAR2(64)          := 'groupcre';
GRP_UPD    CONSTANT   VARCHAR2(64)          := 'groupmod';
GRP_DEL    CONSTANT   VARCHAR2(64)          := 'groupdel';

DEP_ADD    CONSTANT   VARCHAR2(64)          := 'deptcre';
DEP_UPD    CONSTANT   VARCHAR2(64)          := 'deptmod';
DEP_DEL    CONSTANT   VARCHAR2(64)          := 'deptdel';

CLS_ADD    CONSTANT   VARCHAR2(64)          := 'classcre';
CLS_UPD    CONSTANT   VARCHAR2(64)          := 'classmod';
CLS_DEL    CONSTANT   VARCHAR2(64)          := 'classdel';

SUB_ADD    CONSTANT   VARCHAR2(64)          := 'subclasscre';
SUB_UPD    CONSTANT   VARCHAR2(64)          := 'subclassmod';
SUB_DEL    CONSTANT   VARCHAR2(64)          := 'subclassdel';
```

Package Body – Global Variables

```
  LP_seq_no       MERCHHIER_MFQUEUE.SEQ_NO%TYPE  := NULL;
    LP_error_status VARCHAR2(1)                    := NULL;

   cursor C_QUEUE( P_thread_val in number) is
      select q.rowid,
             q.seq_no,
             q.division,
             q.group_no,
             q.dept,
             q.class,
             q.subclass,
             q.div_name,
             q.buyer,
             q.merch,
             q.total_market_amount,
             q.group_name,
             q.dept_name,
             q.profit_calc_type,
             q.purchase_type,
             q.bud_int,
             q.bud_mkup,
             q.markup_calc_type,
             q.otb_calc_type,
             q.dept_vat_incl_ind,
             q.class_name,
             q.class_vat_ind,
             q.subclass_name,
             q.message_type,
             q.pub_status
        from merchhier_mfqueue q
       where q.seq_no = nvl(LP_seq_no,(select min(q2.seq_no)
```

```
                                                                  from
merchhier_mfqueue q2
                                                                 where q2.thread_no
= nvl(P_thread_val, q2.thread_no)
                                                                     and
q2.pub_status = 'U'))
           for update NOWAIT;
```

Function Level Description – ADDTOQ

```
Function: ADDTOQ
           (O_error_msg       OUT            VARCHAR2,
            I_message_type    IN             MERCHHIER_MFQUEUE.MESSAGE_TYPE%TYPE,
            I_division        IN             DIVISION.DIVISION%TYPE,
            I_division_rec    IN             DIVISION%ROWTYPE,
            I_group_no        IN             GROUPS.GROUP_NO%TYPE,
            I_groups_rec      IN             GROUPS%ROWTYPE,
            I_dept            IN             DEPS.DEPT%TYPE,
            I_deps_rec        IN             DEPS%ROWTYPE,
            I_class           IN             CLASS.CLASS%TYPE,
            I_class_rec       IN             CLASS%ROWTYPE,
            I_subclass        IN             SUBCLASS.SUBCLASS%TYPE,
            I_subclass_rec    IN             SUBCLASS%ROWTYPE)
```

If multi-threading is being used, call API_LIBRARY.RIB_SETTINGS to get the number of threads used for the publisher. Using the number of threads, and the business object ID, calculate the thread value.

Insert a record into the MERCHHIER_MFQUEUE.

Function Level Description – GETNXT

```
Procedure: GETNXT
           (O_status_code     OUT            VARCHAR2,
            O_error_msg       OUT            VARCHAR2,
            O_message_type    OUT            VARCHAR2,
            O_message         OUT            RIB_OBJECT,
            O_bus_obj_id      OUT            RIB_BUSOBJID_TBL,
            O_routing_info    OUT            RIB_ROUTINGINFO_TBL,
            I_num_threads     IN             NUMBER DEFAULT 1,
            I_thread_val      IN             NUMBER DEFAULT 1)
```

The RIB calls GETNXT to get messages. The procedure will use the C_QUEUE cursor defined in the specification of the package body to find the next message on the MERCHHIER_MFQUEUE to be published to the RIB.

After retrieving a record from the queue table, GETNXT checks for records on the queue with a status of ‘H’ospital. If there are any such records for the current business object, GETNXT should raise an exception to send the current message to the hospital.

The information from the MERCHHIER_MFQUEUE table is passed to PROCESS_QUEUE_RECORD. PROCESS_QUEUE_RECORD will build the Oracle Object message to pass back to the RIB. If PROCESS_QUEUE_RECORD does not run successfully, GETNXT should raise an exception.

After PROCESS_QUEUE_RECORD returns an oracle object to pass to the RIB, this procedure will delete the record on MERCHHIER_MFQUEUE that was just processed.

If any exception is raised in GETNXT, including the exception raised by an unsuccessful call to PROCESS_QUEUE_RECORD, HANDLE_ERRORS should be called.

Function Level Description – PUB_RETRY

```
Procedure: PUB_RETRY
          (O_status_code     OUT            VARCHAR2,
           O_error_msg       OUT            VARCHAR2,
           O_message_type    IN  OUT        VARCHAR2,
           O_message         OUT            RIB_OBJECT,
           O_bus_obj_id      IN  OUT        RIB_BUSOBJID_TBL,
           O_routing_info    IN  OUT        RIB_ROUTINGINFO_TBL,
           I_REF_OBJECT      IN             RIB_OBJECT);
```

Same as GETNXT except:

The record on MERCHHIER_MFQUEUE must match the passed in sequence number (contained in the ROUTING_INFO).

Function Level Description – PROCESS_QUEUE_RECORD (local)

This function controls the building of Oracle Objects given the business transaction's key values and a message type. It contains all of the shared processing between GETNXT and PUB_RETRY.

If the record from MERCHHIER_MFQUEUE table is an add or update (DIV_ADD, DIV_UPD, GRP_ADD, and so on)

- Build the appropriate ref Oracle Object to publish to the RIB.

If the record from MERCHHIER _MFQUEUE table is a delete (DIV_DEL, GRP_DEL, and so on)

- Build the appropriate ref Oracle Object to publish to the RIB.

In addition to building the Oracle Objects, this function will populate the business object ID. If the message is for a division, group or department, the business object ID will be the division, group, or department respectively. If the message is for a class, the business object will be the class and department combination. If the message is for a subclass, the business object ID will be the subclass, class and department combination.

Function Level Description – HANDLE_ERRORS (local)

HANDLE_ERRORS is called from GETNXT and PUB_RETRY when an exception is raised.

If the error is a non-fatal error, GETNXT passes the sequence number of the driving MERCHHIER_MFQUEUE record back to the RIB in the ROUTING_INFO. It sends back a status of 'H'ospital to the RIB as well. It then updates the status of the queue record to 'H'ospital so that it will not get picked up again by the driving cursor in GETNXT.

If the error is a fatal error, a status of 'E'rror is returned to the RIB.

The error is considered non-fatal if no DML has occurred yet. Whenever DML has occurred, then the global variable LP_error_status is flipped from 'H'ospital to 'E'rror.

Trigger Impact

Trigger Name: EC_TABLE_DIV_AIUDR.TRG

Trigger File Name: ec_table_div_aiudr.trg

Table: DIVISION

Create a trigger on the DIVISION table to capture inserts, updates, and deletes.

Inserts

- Send the appropriate column values to the ADDTOQ procedure in the MFM with the message type RMSMFM_FAMILY.DIV_ADD.

Updates

- Send the appropriate column values to the ADDTOQ procedure in the MFM with the message type RMSMFM_FAMILY.DIV_UPD.

Deletes

- Send the appropriate column values to the ADDTOQ procedure in the MFM with the message type RMSMFM_FAMILY.DIV_DEL.

Trigger Name: EC_TABLE_GRO_AIUDR.TRG

Trigger File Name: ec_table_gro_aiudr.trg

Table: GROUPS

Create a trigger on the GROUPS table to capture inserts, updates, and deletes.

Inserts

- Send the appropriate column values to the ADDTOQ procedure in the MFM with the message type RMSMFM_FAMILY.GRP_ADD.

Updates

- Send the appropriate column values to the ADDTOQ procedure in the MFM with the message type RMSMFM_FAMILY.GRP_UPD.

Deletes

- Send the appropriate column values to the ADDTOQ procedure in the MFM with the message type RMSMFM_FAMILY.GRP_DEL.

Trigger Name: EC_TABLE_DEP_AIUDR.TRG

Trigger File Name: ec_table_dep_aiudr.trg

Table: DEPS

Create a trigger on the DEPS table to capture inserts, updates, and deletes.

Inserts

- Send the appropriate column values to the ADDTOQ procedure in the MFM with the message type RMSMFM_FAMILY.DEP_ADD.

Updates

- Send the appropriate column values to the ADDTOQ procedure in the MFM with the message type RMSMFM_FAMILY.DEP_UPD.

Deletes

- Send the appropriate column values to the ADDTOQ procedure in the MFM with the message type RMSMFM_FAMILY.DEP_DEL.

Trigger Name: EC_TABLE_CLA_AIUDR.TRG

Trigger File Name: ec_table_cla_aiudr.trg

Table: CLASS

Create a trigger on the CLASS table to capture Inserts, Updates, and Deletes.

Inserts

- Send the appropriate column values to the ADDTOQ procedure in the MFM with the message type RMSMFM_FAMILY.CLS_ADD.

Updates

- Send the appropriate column values to the ADDTOQ procedure in the MFM with the message type RMSMFM_FAMILY.CLS_UPD.

Deletes

- Send the appropriate column values to the ADDTOQ procedure in the MFM with the message type RMSMFM_FAMILY.CLS_DEL.

Trigger Name: EC_TABLE_SCL_AIUDR.TRG

Trigger File Name: ec_table_scl_aiudr.trg

Table: SUBCLASS

Create a trigger on the SUBCLASS table to capture inserts, updates, and deletes.

Inserts

- Send the appropriate column values to the ADDTOQ procedure in the MFM with the message type RMSMFM_FAMILY.SUB_ADD.

Updates

- Send the appropriate column values to the ADDTOQ procedure in the MFM with the message type RMSMFM_FAMILY.SUB_UPD.

Deletes

- Send the appropriate column values to the ADDTOQ procedure in the MFM with the message type RMSMFM_FAMILY.SUB_DEL.

DTD

Here are the filenames that correspond with each message type. Please consult the RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
divisoncre	Division Create Message	MrchHrDivDesc.dtd
divisonmod	Division Modify Message	MrchHrDivDesc.dtd
divisiondel	Division Delete Message	MrchHrDivRef.dtd
groupcre	Group Detail Create Message	MrchHrGrpDesc.dtd
groupmod	Group Detail Modify Message	MrchHrGrpDesc.dtd
groupdel	Group Detail Delete Message	MrchHrGrpRef.dtd
deptcre	Department Detail Create Message	MrchHrDeptDesc.dtd
deptmod	Department Detail Modify Message	MrchHrDeptDesc.dtd
deptdel	Department Detail Delete Message	MrchHrDeptRef.dtd
classcre	Class Detail Create Message	MrchHrClsDesc.dtd
classmod	Class Detail Modify Message	MrchHrClsDesc.dtd
classdel	Class Detail Delete Message	MrchHrClsRef.dtd
subclasscre	Subclass Detail Create Message	MrchHrSclsDesc.dtd
subclassmod	Subclass Detail Modify Message	MrchHrSclstDesc.dtd
subclassdel	Subclass Detail Delete Message	MrchHrSclsRef.dtd

Table Impact

TABLE	SELECT	INSERT	UPDATE	DELETE
MERCHHIER_MFQUEUE	Yes	Yes	Yes	Yes
DIVISION	Yes	No	No	No
DEPT	Yes	No	No	No
CLASS	Yes	No	No	No
SUBCLASS	Yes	No	No	No

Assumptions

Delay all DML statements as late as possible. Once DML statements have taken place, any error becomes a fatal error rather than a hospital error.

Order

Business Overview

Purchase order (PO) functionality in RMS consists of order messages published to the Oracle Retail Integration Bus (RIB), and batch modules that internally process purchase order data and upload EDI transmitted orders. This overview describes how both order messages and batch programs process this data.

How Purchase Orders are Created

Purchase orders are created:

- Online, through the ordering dialog
- Automatically, through replenishment processes
- Against a supplier contract type 'B'
- By a supplier, in a vendor managed inventory environment
- Through direct store delivery (defined as delivery of merchandise or a service that does not result from the prior creation of a PO). For more information, see the chapter "Purchase Order Batch" in volume 1 of this RMS Operations Guide.
- Through the Buyer Worksheet dialog
- Through truck splitting

For more information about the replenishment order building process, see the chapter "Replenishment Batch" in volume 1 of this RMS Operations Guide.

Purchase Order Messages

After purchase orders are published to the RIB, the following associated activities can occur:

- Work orders associated with items on the PO are published to the RIB through the work order message process.
- An allocation (also known as pre-distribution) of items on the PO is published to the RIB through the stock order message process.
- A PO can be closed only after all appointments against the purchase order are closed. A closed appointment indicates that all merchandise has been received. RMS subscribes to appointment messages from the RIB. For more information, see the section 'Appointments' in the chapter "Subscription Design" in this volume of the RMS Operations Guide.
- 'Version' refers to any change to a purchase order by a retailer's buyer; whereas 'Revision' refers to any change to a purchase order initiated by a supplier.

Order Message Processes

RMS publishes two sets of PO messages to the RIB for two kinds of subscribing applications. The first set of messages represents only virtual locations in RMS. Virtual locations exist whenever the retailer runs RMS in a multi-channel environment. Applications that understand virtual locations subscribe to these messages.

RMS publishes a second set of PO messages for applications that can subscribe only to conventional, physical location data, such as a warehouse management system. One or both subscribing methods (virtual locations and physical locations) can be used in a multi-channel environment. In a single-channel environment, both sets of messages are identical.

Ordering publication will be primarily based off of the ORDHEAD, ORDSKU, and ORDLOC tables.

ORDHEAD is the parent table containing high level ordering information such as what supplier is being ordered from, when the order should take place, and so on. ORDSKU is a child of ORDHEAD and contains the item(s) that are ordered, the size of the pack being ordered, and so on.

ORDLOC is a child of ORDSKU and contains the location(s) each item on the order is going to and how much of each item is ordered. Based on this table hierarchy, two levels of messages will exist for order publishing. A header message which is primarily driven off of the ORDHEAD table, and a detail message which is primarily driven off both the ORDSKU and ORDLOC tables.

Each message level will contain three types of messages; create, modify, and delete. The ‘POCre’ or ‘POHdrMod’ message will be created when an insertion or modification to the ORDHEAD table is made respectively. The ‘PODel’ message will be created when an order is deleted from the ORDHEAD table. ‘PODtlCre’ or ‘PODtlMod’ message will be created when a record is inserted or modified on the ORDLOC table respectively. ‘PODtlDel’ will be created when an ORDLOC record is deleted.

Functionality Checklist

Description	RMS	RIB
RMS must publish order information		
Create new Publisher	X	X

Form Impact

None.

Business Object Records

Create the following record types in the rmsmfm_order package specification:

```
TYPE ordhead_msg_rectype IS RECORD
               (ordhead_rec                ORDHEAD%ROWTYPE,
                doc_type                   VARCHAR2(1),
                order_type_desc            CODE_DETAIL.CODE_DESC%TYPE,
                dept_name                  DEPS.DEPT_NAME%TYPE,
                buyer_name                 BUYER.BUYER_NAME%TYPE,
                promotion_desc             VARCHAR2(160),
                terms_code                 TERMS.TERMS_CODE%TYPE,
                payment_method_desc        CODE_DETAIL.CODE_DESC%TYPE,
                backhaul_type_desc         CODE_DETAIL.CODE_DESC%TYPE,
                ship_method_desc           CODE_DETAIL.CODE_DESC%TYPE,
                purchase_type_desc         CODE_DETAIL.CODE_DESC%TYPE,
                ship_pay_method_desc       CODE_DETAIL.CODE_DESC%TYPE,
                fob_trans_res_code_desc    CODE_DETAIL.CODE_DESC%TYPE,
                fob_title_pass_code_desc   CODE_DETAIL.CODE_DESC%TYPE,
                factory_desc               PARTNER.PARTNER_DESC%TYPE,
                agent_desc                 PARTNER.PARTNER_DESC%TYPE,
                discharge_port_desc        OUTLOC.OUTLOC_DESC%TYPE,
                lading_port_desc           OUTLOC.OUTLOC_DESC%TYPE,
                po_type_desc               PO_TYPE.PO_TYPE_DESC%TYPE);
```

Package Impact

Create a Worksheet Order

1. Prerequisites: Orders can be created through various methods. Orders can be created manually by a user, through a replenishment process (order can be created in either worksheet or approved status), uploaded from a vendor, or through a contract.
2. Activity Detail: At this point, the order is not seen externally from RMS.
3. Messages: When the order is created, a header message ‘POCre’ is written to the ordering queue table. Upon detail additions, each will have a ‘PODtlCre’ message written to the ordering queue. Ordering messages are added, updated, and removed from the queue as the order is modified prior to approval.

Modify Pre-Approved

1. Prerequisites: Order is still in worksheet status and has not been approved and set back to worksheet.
2. Activity Detail: At this point, items can be modified, added or removed from the order. The order can be split, scaled, and rounded in addition to having deals, brackets applied.
3. Messages: Each change will cause a ‘POHdrMod’ or ‘PODtlMod’ message. These messages will replace previous create messages if there was a modification, delete a previous message if there was a delete, or add a new message to the queue for inserts.

Approve

1. Prerequisites: Line items must exist for the order to be approved. Relevant dates (not before, not after, pickup) must exist, plus certain other business validation rules based on system options.
2. Activity Detail: At this point, the order is initially approved which means external systems will now have constant visibility to all ordering transactions. The user can no longer delete line items: Instead, they are cancelled. Canceling decrements the order quantity by amount already received.
3. Messages: The approval message sets an indicator signifying the approval create message should be built. This is a hierarchical snapshot synchronous message built in the family manager by attaching all of the 'PODtlDesc' messages with the 'POHdrDesc' message to create a 'POCre' message.

Modify in 'A' status

1. Prerequisites: Order must be currently approved.
2. Activity Detail: Numerous fields at the header level (none at the detail level) can be changed while the order is approved. This change will create a message.
3. Messages: A 'POHdrMod' message will be created for order at the end of the session the order was modified. This message will be published immediately as the order will already have been published. If the order has not been published, then this message will follow the create message sent out.

Redistribute

1. Prerequisites: Order must be in approved or worksheet status. Order must not be a contract order. No shipments/appointments may exist against the order. Items with allocations cannot be redistributed.
2. Activity Detail: User chooses which items to redistribute. Each chosen details are removed from the order. This will create delete messages for each one. A new location is then chosen to redistribute the items to. Each item/location record will create a message. Note that if user chooses to redistribute records, then cancels out of redistribution, delete and create messages for the chosen records will be inserted into the queue even though no changes were actually made online.
3. Messages: A 'PODtlDel' message is created for each item/location removed from the order. If the order has not yet been approved, then these messages will remove previous create messages. For already approved orders, then a message will be published. For each redistributed item, a 'PODtlCre' message will be created.

Unapprove

1. Prerequisites: Order must currently be in approved status. Shipments/Appoinments may exist against the order.
2. Activity Detail: This will change the status of the order back to worksheet. This will create a message. Existing details will be modifiable. New records may be added to the order. Items may not be deleted from the order. However, the order quantity of the items can be canceled down to the received or appointment expected quantity.
3. Messages: A 'POHdrMod' message will be created for order at the end of the session the order was modified. This message will be published immediately as the order will already have been published. If the order has not been published, then this message will follow the create message sent out.

Modify

1. Prerequisites: Order must be in worksheet status and have already been approved.
2. Activity Detail: If modifications occur at the header level, a header message will be created. A detail message will be created for each modified or added detail record. Detail records cannot be deleted; only their quantities can be canceled.
3. Message: A ‘POHdrMod’ message will be created for order at the end of the session if the header was modified. A ‘PODtlCre’ or ‘PODtlMod’ message will be created for each detail record added or modified respectively.

Close

1. Prerequisites: Order must currently be in approved status or in worksheet status and have been previously approved. No outstanding shipments/appointments may exist against any line items of the order.
2. Activity Detail: The status will change to closed. This will create a message. Any outstanding unreceived quantities will be canceled out. No details will be modifiable while the order is in this status.
3. Message: A ‘POHdrMod’ message will be created for order at the end of the session the order was modified. A ‘PODtlCre’ message will be created for each line item that had outstanding un-received quantity. These messages will be published immediately as the order will already have been published. If the order has not been published, then this message will follow the create message sent out.

Reinstate

1. Prerequisites: Order must be in closed status. Orders that have been fully received (closed through receiving dialogue) cannot be reinstated.
2. Activity Detail: The status will change to worksheet. This will create a header level message. All canceled quantities will be added back to order quantities. Details will be modifiable.
3. Message: A ‘POHdrMod’ message will be created for order at the end of the session the order was modified. A ‘PODtlMod’ message will be created for each line item that had outstanding canceled quantity. These messages will be published immediately as the order will already have been published. If the order has not been published, then this message will follow the create message sent out.

Delete

1. Prerequisites: If the user deletes the order manually, then the order needs to be in worksheet status and never been approved. Else, for approved orders, the following explanation details the business validation for deleting orders. If the import indicator on the SYSTEM OPTIONS table (import_ind) is 'N' and if invoice matching is not installed, then all details associated with an order are deleted when the order has been closed for more months than specified in UNIT_OPTIONS (order_history_months). If invoice matching is installed, then all details associated with an order are deleted when the order has been closed for more months than specified in UNIT_OPTIONS (order_history_months). Orders are deleted only if shipments from the order have been completely matched to invoices or closed, and all those invoices have been posted. If the import indicator on the SYSTEM OPTIONS table (import_ind) is 'Y' and if invoice matching is not installed, then all details associated with the order are deleted when the order has been closed for more months than specified in UNIT_OPTIONS (order_history_months), as long as all ALC records associated with an order are in 'Processed' status, specified in ALC_HEAD (status). If invoice matching is installed, then all details associated with an order are deleted when the order has been closed for more months than specified in UNIT_OPTIONS (order_history_months), as long as all ALC records associated with an order are in 'Processed' status, specified in ALC_HEAD (status), and as long as all shipments from the order have been completely matched to invoices or closed, and all those invoices have been posted.
2. Activity Detail: Deleting orders will create a message for each detail attached to the order plus the header record.
3. Messages: If the order has not been approved, then the ‘PODel’ and ‘PODtlDel’ messages created will remove all the previous messages on the ordering queue table. If the order has been approved, then a ‘PODtlDel’ message will be created for each detail record and a ‘PODel’ message for the header.

Package Name: RMSMFM_ORDER

Spec File Name: rmsmfm_orders.pls

Body File Name: rmsmfm_orderb.pls

Package Specification – Global Variables

```
FAMILY      CONSTANT    RIB_SETTINGS.FAMILY%TYPE      ‘order’;
HDR_ADD     CONSTANT    VARCHAR2(64)                  ‘POcre’;
HDR_UPD     CONSTANT    VARCHAR2(64)                  ‘POHdrMod’;
HDR_DEL     CONSTANT    VARCHAR2(64)                  ‘PODel’;
DTL_ADD     CONSTANT    VARCHAR2(64)                  ‘PODtlCre’;
DTL_UPD     CONSTANT    VARCHAR2(64)                  ‘PODtlMod’;
DTL_DEL     CONSTANT    VARCHAR2(64)                  ‘PODtldel’;
```

Function Level Description – ADDTOQ

```
Function: ADDTOQ
          (O_error_message          OUT    VARCHAR2,
           I_message_type           IN     ORDER_MFQUEUE.MESSAGE_TYPE%TYPE,
           I_order_no               IN     ORDHEAD.ORDER_NO%TYPE,
           I_order_type             IN     ORDHEAD.ORDER_TYPE%TYPE,
           I_order_header_status    IN     ORDHEAD.STATUS%TYPE,
           I_supplier               IN     ORDHEAD.SUPPLIER%TYPE,
           I_item                   IN     ORDLOC.ITEM%TYPE,
           I_location               IN     ORDLOC.LOCATION%TYPE,
           I_loc_type               IN     ORDLOC.LOC_TYPE%TYPE,
           I_physical_location      IN     ORDLOC.LOCATION%TYPE)
```

This procedure is called by either the ORDHEAD or ORDLOC row trigger, and takes the message type, table primary key values (order_no for ORDHEAD table and order_no, item, location(virtual) and physical location for ORDLOC table) and the message itself. It inserts a row into the message family queue along with the passed in values and the next sequence number from the message family sequence. The pub status will always be 'U' except for PO create messages, then it will be 'N'. The approve indicator will always be 'N' except when the order is approved for the first time, then it will be 'Y'. It returns error codes and strings according to the standards of the application in which it is being implemented.

Function Level Description – GETNXT

```
Procedure: GETNXT
              (O_status_code        OUT    VARCHAR2,
               O_error_msg          OUT    VARCHAR2,
               O_message_type       OUT    VARCHAR2,
               O_message            OUT    RIB_OBJECT,
               O_bus_obj_id         OUT    RIB_BUSOBJID_TBL,
               O_routing_info       OUT    RIB_ROUTINGINFO_TBL,
               I_num_threads        IN     NUMBER DEFAULT 1,
               I_thread_val         IN     NUMBER DEFAULT 1)
```

This publicly exposed procedure is typically called by a RIB publication adaptor. Its parameters are well defined and arranged in a specific order. The message type is the RIB defined short message name, the message is the xml message, and the family key(s) (order_no for ORDHEAD table and order_no, item, location(virtual) and physical location for ORDLOC table) are the key for the message as pertains to the family, not all of which will necessarily be populated for all message types.

This program loops through each message on the ORDER_MFQUEUE table, and calls PROCESS_QUEUE_RECORD. When no messages are found, the program exits returning the 'N'o message found API code.

The error text parameter contains application-generated information, such as the application's sequence number of the message that failed, and the Oracle or other error that occurred when the retrieval failed.

Function Level Description – PUB_RETRY

```
Procedure: PUB_RETRY
              (O_status_code        OUT          VARCHAR2,
               O_error_msg          OUT          VARCHAR2,
               O_message_type       IN OUT       VARCHAR2,
               O_message            OUT          RIB_OBJECT,
               O_bus_obj_id         IN OUT       RIB_BUSOBJID_TBL,
               O_routing_info       IN OUT       RIB_ROUTINGINFO_TBL,
               I_REF_OBJECT         IN           RIB_OBJECT);
```

Same as GETNXT except:

It only loops for a specific row in the ORDER_MFQUEUE table. The record on ORDER_MFQUEUE must match the passed in sequence number (contained in the ROUTING_INFO).

Function Level Description – PROCESS_QUEUE_RECORD (local)

This function controls the building of Oracle Objects given the business transaction's key values and a message type. It contains all of the shared processing between GETNXT and PUB_RETRY.

Check to see if the business object is being published for the first time. If the published_ind on the pub_info table is 'N', then it is not yet published.

If the record from ORDER_MFQUEUE table is a header delete (HDR_DEL) and published_ind is 'N'

- Delete the record from the pub info table.
- Call DELETE_QUEUE_REC.

If the record from ORDER_MFQUEUE table is a header delete (HDR_DEL)

- Build and pass the RIB_PORef_REC object.
- Call GET_ROUTING_TO_LOCS
- Delete the record from the pub info table.
- Delete the record from the order_details_published table
- Call DELETE_QUEUE_REC.

If the published_ind is 'N' or 'I'

- If the publish_ind is 'N' call MAKE_CREATE with the message_type 'HDR_ADD'.
- Otherwise, call MAKE_CREATE with the message_type 'DTL_ADD'.

If the record from ORDER_MFQUEUE table is a header update (HDR_UPD)

- Call BUILD_HEADER_OBJECT
- Update order_pub_info by setting the published indicator to 'Y'
- Call GET_ROUTING_TO_LOCS
- Call DELETE_QUEUE_REC

If the record from ORDER_MFQUEUE table is a detail insert (DTL_ADD) or detail update (DTL_UPD)

- Call BUILD_DETAIL_CHANGE_OBJECTS

If the record from ORDER_MFQUEUE table is a detail delete (DTL_DEL)

- Call BUILD_DETAIL_DELETE
- Call ROUTING_INFO_ADD

Function Level Description – MAKE_CREATE (local)

This function is used to create the Oracle Object for the initial publication of a business transaction.

- Call BUILD_HEADER_OBJECT to get a header level Oracle Object plus any extra functional holders.
- Call BUILD_DETAIL_OBJECTS to get a table of detail level Oracle objects and a table of ORDER_MFQUEUE rowids to delete.
- Use the header level Oracle Object and functional holders to update the ORDER_PUB_INFO.
- Delete records from the ORDER_MFQUEUE for all rowids returned by BUILD_DETAIL_OBJECTS. Deletes are done by rowids instead of business transaction keys to ensure that nothing is deleted off the queue that has not been published.
- If the entire business transaction was added to the Oracle Object, also delete the ORDER_MFQUEUE record that was picked up by GETNXT. If the entire business transaction was not published the system needs to leave something on the ORDER_MFQUEUE to ensure that the rest of it is picked up by the next call to GETNXT.
- The header and detail level Oracle Objects are combined and returned.

Function Level Description – BUILD_HEADER_OBJECT (local)

Accepts header key values, performs necessary lookups, builds and returns a header level Oracle Object.

Call GET_MSG_HEADER.

Function Level Description – BUILD_DETAIL_OBJECTS (local)

The function is responsible for building detail level Oracle Objects. It builds as many detail Oracle Object as it can given the passed in message type and business object keys.

If the function is being called from MAKE_CREATE:

- Select any unpublished detail records from the business transaction (use an indicator on the functional detail table itself or ORDER_DETAILS_PUBLISHED). Create Oracle Objects for details that are selected by calling BUILD_SINGLE_DETAIL.

If the function is not being called from MAKE_CREATE:

- Select any details on the ORDER_DETAILS_PUBLISHED that are for the same business transaction and for the same message type. Create Oracle Objects for details that are selected by calling BUILD_SINGLE_DETAIL.

Create other necessary Oracle objects and insert into and update the ORDER_DETAILS_PUBLISHED table for details that were published.

Function Level Description – BUILD_SINGLE_DETAIL (local)

Accept inputs and build a detail level Oracle Object. Perform any lookups needed to complete the Oracle Object.

Function Level Description – BUILD_DETAIL_CHANGE_OBJECTS (local)

Either pass in a header level Oracle Object or call BUILD_HEADER_OBJECT to build one.

Call BUILD_SINGLE_DETAIL to get the delete level Oracle Objects.

Perform any BULK DML statements given the output from BUILD_ DETAIL_OBJECTS

Build any ROUTING_INFO as needed.

Function Level Description – BUILD_DETAIL_DELETE (local)

Either pass in a header level ref Oracle Object or build a header level ref Oracle Object.

Perform a cursor for loop on ORDER_MFQUEUE and build as many detail ref Oracle Objects as possible without exceeding the MAX_DETAILS_TO_PUBLISH.

Perform any BULK DML statements for deletion from ORDER_MFQUEUE and ORDER_DETAILS_PUBLISHED.

Call BUILD_DETAIL_DELETE_WH for Warehouses.

Function Level Description – DELETE_QUEUE_REC (local)

Delete the passed in data from the queue table.

Function Level Description – BUILD_DETAIL_DELETE_WH (local)

Builds Oracle objects based on the records found in the queue table that are from the ORDLOC table.

Function Level Description – ROUTING_INFO_ADD (local)

Build any ROUTING_INFO.

Function Level Description – GET_ROUTING_TO_LOCS (local)

Build the ROUTING_INFO by adding locations.

Function Level Description – GET_MSG_HEADER (local)

Perform any lookups to complete the header information.

Function Level Description – LOCK_THE_BLOCK (local)

This function locks all queue records for the current business object. This is to ensure that GETNXT does not wait on any business processes that currently have the queue table locked and have not committed. This can occur because ADDTOQ, which is called from the triggers, deletes from the queue table for DTL_UPD, DTL_DEL, and HDR_DEL messages.

Function Level Description – HANDLE_ERRORS (local)

```
PROCEDURE HANDLE_ERRORS
       (O_status_code         IN OUT         VARCHAR2,
        O_error_message       IN OUT         VARCHAR2,
        O_message             IN OUT         nocopy RIB_OBJECT,
        O_bus_obj_id          IN OUT         nocopy RIB_BUSOBJID_TBL,
        O_routing_info        IN OUT         nocopy RIB_ROUTINGINFO_TBL,
        I_seq_no              IN             order_mfqueue.seq_no%TYPE,
        I_order_no            IN             order_mfqueue.order_no%TYPE,
        I_item                IN             order_mfqueue.item%TYPE,
        I_physical_location   IN             order_mfqueue.physical_location%TYPE,
        I_loc_type            IN             order_mfqueue.loc_type%TYPE)
```

HANDLE_ERRORS is called from GETNXT and PUB_RETRY when an exception is raised.

If the error is a non-fatal error, GETNXT passes the sequence number of the driving ORDER_MFQUEUE record back to the RIB in the ROUTING_INFO. It sends back a status of ‘H’ospital to the RIB as well. It then updates the status of the queue record to ‘H’ospital, so that it will not get picked up again by the driving cursor in GETNXT.

If the error is a fatal error, a status of ‘E’rror is returned to the RIB.

The error is considered non-fatal if no DML has occurred yet. Whenever DML has occurred, then the global variable LP_error_status is flipped from ‘H’ospital to ‘E’rror.

Trigger Impact

Create a trigger on the ORDHEAD and ORDLOC to capture inserts, updates, and deletes.

Trigger Name: EC_TABLE_OHE_AIUDR.TRG

Trigger File Name: ec_table_ohe_aiudr.trg

Table: ORDHEAD

This trigger fires when an ORDHEAD record has been inserted, updated or deleted on any of the columns published. Each action is detailed below. In general, this trigger calls RMSMFM_ORDER.ADDTOQ to place the message and order onto the ORDER_MFQUEUE table.

Inserts

- Send the header level info to the ADDTOQ procedure in the MFM with the message type HDR_ADD.

Updates

- Send the header level info to the ADDTOQ procedure in the MFM with the message type HDR_UPD.

Deletes

- Send the header level info to the ADDTOQ procedure in the MFM with the message type HDR_DEL.

Trigger Name: EC_TABLE_OLO_AIUDR.TRG

Trigger File Name: ec_table_olo_aiudr.trg

Table: ORDLOC

This triggers fires when an ORDLOC record has been inserted, updated or deleted on the qty_ordered or estimated_instock_date columns. Each action is detailed below. In general, this trigger calls RMSMFM_ORDER.ADDTOQ to place the message and order, item, location onto the ORDER_MFQUEUE table.

Inserts

- Send the detail level info to the ADDTOQ procedure in the MFM with the message type DTL_ADD.

Updates

- Send the header level info to the ADDTOQ procedure in the MFM with the message type DTL_UPD.

Deletes

- Send the header level info to the ADDTOQ procedure in the MFM with the message type DTL_DEL.

DTD

Here are the filenames that correspond with each message type. Please consult the RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
POCre	Purchase Order Create Message	PODesc.dtd
POHdrMod	Purchase Order Modify Message	PODesc.dtd
PODel	Purchase Order Delete Message	PORef.dtd
PODtlCre	Purchase Order Detail Create Message	PODesc.dtd
PODtlMod	Purchase Order Detail Modify Message	PODesc.dtd
PODtlDel	Purchase Order Detail Delete Message	PORef.dtd

Table Impact

TABLE	SELECT	INSERT	UPDATE	DELETE
ORDHEAD	Yes	No	No	No
ORDLOC	Yes	No	No	No
ORDSKU	Yes	No	No	No
ORDER_MFQUEUE	Yes	Yes	Yes	Yes
ORDER_PUB_INFO	Yes	Yes	Yes	Yes
ORDER_DETAILS_PUBLISHED	Yes	Yes	Yes	Yes

Design Assumptions

- One of the primary assumptions in the current approach is that ease of code will outweigh performance considerations. It is hoped that the ‘trickle’ nature of the flow of data will decrease the need to dwell on performance issues and instead allow developers to code in the easiest and most straight forward manner.
- The adaptor is only setup to call stored procedures, not stored functions. Any public program then needs to be a procedure.

Partner

Business Overview

RMS publishes data about partners in messages to the Oracle Retail Integration Bus (RIB). Other applications that need to keep their partners synchronized with RMS subscribe to these messages.

Using the 'External Finisher' functionality, a retailer can send all/any goods to an external location for being repaired or worked upon. The goods can be at a warehouse or a store and sent to an external location for being worked upon and then transferred back to either location. For example, if a retailer wants to have a partner add embroidery to a shirt, then the shirt will be transferred from the original warehouse, to the partner, and then sent on from the partner to the receiving store. The RIB helps to coordinate this partnership activity.

External Finishers

External finishers are created as partners in RMS, and given the Partner Type 'E', indicating that the partner is an External finisher. Once a new external finisher is set up in RMS, a trigger on the partner table adds the external finisher to a new queue table. Information on that table is published via the RIB. A conversion of this RIB message converts the external finisher to a 'Location' so that it can be consumed by the location APIs of external systems such as RWMS.

RWMS and other integration subsystems subscribe to the external finisher through their location subscription APIs. A RIB TAFR parses the partner messages of partner type 'E' and returns location attributes for RWMS and other integration subsystems to subscribe to. RMS ensures that there will never be duplicates among the partner ID, store ID and warehouse ID.

The RWMS transfer subscription process does not check for location types. As a result, transfers involving an external finisher are treated like any other location types.

To facilitate the routing of external finisher and primary address of the primary address type, header level routing info will contain the name of 'partner_type' with value 'E'. Detail level routing info will contain the name of 'primary_addr_type_ind' with value of 'Y' or 'N' and the name of 'primary_addr_ind' with value of 'Y' or 'N'. This will allow the RIB to route the external finishers and their addresses to the correct applications.

RMS will publish the create, mod, and delete messages of partners, along with their multiple addresses, to the RIB via a partner publishing message.

The insert/update/delete on the partner table and the addr table with module 'PTNR' (for partner) will be published. The output message will be in hierarchical structure, with partner information at the header level and the address information at the detail level. Because this is a low volume publisher, multi-threading capability is not supported. In addition, the system assumes that it only needs to publish the current state of the partner, not every change.

If multiple addresses are associated with a partner, this publisher is designed with the assumption that RWMS and other integration subsystems only subscribe to the primary address of the primary address type.

Functionality Checklist

Description	RMS	RIB
RMS must publish Partner information		
Create new publisher Partner	X	X

Form Impact

None.

Business Object Records

Create the following business objects to assist the publishing process:

Create a PARTNER_KEY_REC (in rmsmfm_partners.pls) that contains the functional keys to partner_mfqueue publishing:

```
TYPE partner_key_rec IS RECORD
               (PARTNER_TYPE         PARTNER.PARTNER_TYPE%TYPE,
               PARTNER_ID            PARTNER.PARTNER_ID%TYPE,
               ADDR_KEY              ADDR.ADDR_KEY%TYPE);  -- optional
```

Package Impact

Business Object ID

The business object ID for partner publisher is partner type and partner ID, which uniquely identifies a partner for publishing. The RIB uses the business object ID to determine message dependencies when sending messages to a subscribing application. If a Create message has already failed in the subscribing application, and a Modify/Delete message is about to be sent from the RIB to the subscribing application, the RIB will not send the Modify/Delete message if it has the same business object ID as the failed Create message. Instead, the Modify/Delete message will go directly to the hospital.

For example:

```
Partner type X, partner A, message type ‘PartnerCre’ fails in subscriber
Partner type X, partner B, message type ‘PartnerCre’ processes successfully in subscriber
Partner type X, partner A, message type ‘PartnerMod’ goes directly from RIB to hospital.
Partner type X, partner B, message type ‘PartnerMod’ goes from RIB to subscriber.
Partner type X, partner A, message type ‘PartnerDel’ goes directly from RIB to hospital.
```

For Header Insert,Update, and Delete:

HDR_ADD

- Create a partner with 1 or more addresses. HDR_ADD message will be added to the queue.

HDR_UPD

- Update an existing partner header record that has already been published. HDR_UPD message will be added to the queue.
- Update an existing partner header record that has **not** been published yet. HDR_UPD message should **not** be added to the queue. When the partner header is published, it will fetch the latest information on PARTNER.

HDR_DEL

- Delete an existing partner header record that has already been published. HDR_DEL message must be added to the queue.

- Delete an existing partner header record that has **not** been published yet. HDR_DEL message should **not** be added to the queue. In addition, all other message related to the partner must be deleted from the queue. The partner must also be deleted from PARTNER_PUB_INFO.

For Detail Insert, Update, and Delete:

DTL_ADD

- Add a new address to an existing partner. DTL_ADD message must be added to the queue.

DTL _UPD

- Update an existing address of a partner that has already been published. Any existing DTL_UPD message for the partner/address in the queue must be deleted. The current DTL_UPD message for the partner/address must be added to the queue.
- Update an existing address of a partner that has NOT been published yet. Any existing DTL_UPD message for the partner/address in the queue must be deleted. The current DTL_UPD message for the partner/address does **not** need to be added to the queue. This is because when the new address is published, it will fetch the latest information on ADDR table.

DTL _DEL

- Delete an existing address of a partner that has already been published. The address has also been published. Any existing message for the partner/address in the queue must be deleted. The current DTL_DEL for the partner/address must be added to the queue.
- Delete an existing address of a partner that has already been published. The address has **not** been published yet. Any existing message for the partner/address in the queue must be deleted. The current DTL_DEL for the partner/address does **not** need to be added to the queue.

Package Name: RMSMFM_PARTNER

Spec File Name: rmsmfm_partners.pls

Body File Name: rmsmfm_partnerb.pls

Package Specification – Global Variables

```
FAMILY      CONSTANT   RIB_SETTINGS.FAMILY%TYPE      'PARTNER';
HDR_ADD     CONSTANT   VARCHAR2(15)                  'partnercre';
HDR_UPD     CONSTANT   VARCHAR2(15)                  'partnermod';
HDR_DEL     CONSTANT   VARCHAR2(15)                  'partnerdel';
DTL_ADD     CONSTANT   VARCHAR2(15)                  'partnerdtlcre';
DTL_UPD     CONSTANT   VARCHAR2(15)                  'partnerdtlmod';
DTL_DEL     CONSTANT   VARCHAR2(15)                  'partnerdtldel';
```

Function Level Description – ADDTOQ

```
Function: ADDTOQ
              (O_error_mesage       OUT     VARCHAR2,
               I_message_type       IN      VARCHAR2,
               I_functional_keys    IN      PARTNER_KEY_REC)
```

This public function puts a partner message on PARTNER_MFQUEUE for publishing to the RIB. It is called from both partner trigger and address trigger. The I_functional_keys will contain partner_type, partner_id and optionally, addr_key.

There are some tasks relating to streamlining the queue clean up process that need to occur in ADDTOQ. The goal is to have at most one record on the queue for a given partner up until its initial publication.

- For header level insert messages (HDR_ADD), insert a record in the PARTNER_PUB_INFO table. The published flag should be set to 'N'. Because this is a low volume business transaction, the partner publisher will not provide multi-threading capability. Therefore, thread value does **not** need to be calculated.
- For HDR_UPD, DTL_ADD, DTL_UPD, DTL_DEL, messages do **not** need to be added to the queue until the business object (partner type and partner ID) has been initially published (PARTNER_PUB_INFO.published = 'N'). This is because when the partner does get published for the first time (HDR_ADD), the current state of the header and details will be queried and published to the RIB.
- For header level delete messages (HDR_DEL), if the business object (partner type and partner ID) has **not** been initially published (PARTNER_PUB_INFO.published = 'N'), delete the record in PARTNER_PUB_INFO. Otherwise, delete every record in the queue for the business object, and add the delete record to the queue.
- If the business object has been initially published (PARTNER_PUB_INFO.published = 'Y'), for detail level messages deletes (DTL_DEL), the system only needs one (the most recent) record per detail in the PARTNER_MFQUEUE. Delete any previous records that exist on the PARTNER_MFQUEUE for the record that has been passed.
- If the business object has been initially published (PARTNER_PUB_INFO.published = 'Y'), for detail level messages updates (DTL_UPD), the system only needs one DTL_UPD (the most recent) record per detail in the PARTNER_MFQUEUE. Delete any previous DTL_UPD records that exist on the PARTNER_MFQUEUE for the record that has been passed. The system does not want to delete any detail inserts that exist on the queue for the detail. The system needs to ensure subscribers are not passed a detail modification message for a detail that they do not yet have.
- For all message types except header level inserts (HDR_ADD), insert a record into the PARTNER_MFQUEUE. One exception is that if the publish_ind on the detail record table (addr) is 'N', do not add the DTL_DEL message to the queue. The logic here is that if the detail line has never been published before, subscribers will not need to delete the detail line that they do not yet have.

Function Level Description – GETNXT

```
Procedure: GETNXT
              (O_status_code        OUT    VARCHAR2,
               O_error_msg          OUT    VARCHAR2,
               O_message_type       OUT    VARCHAR2,
               O_message            OUT    RIB_OBJECT,
               O_bus_obj_id         OUT    RIB_BUSOBJID_TBL,
               O_routing_info       OUT    RIB_ROUTINGINFO_TBL,
               I_num_threads        IN     NUMBER DEFAULT 1,
               I_thread_val         IN     NUMBER DEFAULT 1)
```

This public procedure is called from the RIB to get the next messages. It performs a cursor loop on the unpublished records on the PARTNER_MFQUEUE table (PUB_STATUS = 'U'). It should only need to execute one loop iteration in most cases. For each record retrieved, GETNXT gets the following:

1. A lock of the queue table for the current business object (partner_type and partner_id). The lock is obtained by calling the function LOCK_THE_BLOCK. If there are any records on the queue for the current business object that are already locked, the current message is skipped and picked up again in the next loop iteration.
2. A check for records on the queue with a status of 'H'ospital. If there are any such records for the current business object, GETNXT raises an exception to send the current message to the Hospital.
3. Get the published indicator from the PARTNER_PUB_INFO table.
4. Call PROCESS_QUEUE_RECORD with the current business object.

The loop will need to execute more than one iteration for the following cases

1. When a header delete message exists on the queue for a business object that has not been initially published. In this case, simply remove the header delete message from the queue and loop again.

   Note: The situation above should not happen very often. ADDTOQ will delete all messages for the business object upon header delete if the business object has not been initially published.)

2. The queue is locked for the current business object. This can occur because ADDTOQ, which is called from the triggers, deletes from the queue table for DTL_UPD, DTL_DEL, and HDR_DEL messages.

The information from the PARTNER_MFQUEUE and PARTNER_PUB_INFO table is passed to PROCESS_QUEUE_RECORD. PROCESS_QUEUE_RECORD will build the Oracle Object message to pass back to the RIB. If PROCESS_QUEUE_RECORD does not run successfully, GETNXT raises an exception.

If any exception is raised in GETNXT, including the exception raised by an unsuccessful call to PROCESS_QUEUE_RECORD, HANDLE_ERRORS is called.

Function Level Description – PUB_RETRY

This public procedure performs the same tasks as GETNXT except that it only loops for a specific row in the PARTNER_MFQUEUE table. The record on PARTNER_MFQUEUE must match the passed in sequence number (contained in the ROUTING_INFO).

Function Level Description – PROCESS_QUEUE_RECORD (local)

This private function controls the building of Oracle Objects (DESC or REF) given the business transaction's key values and a message type. It contains all of the shared processing between GETNXT and PUB_RETRY. Note that the message_type of HDR_ADD can potentially be changed to a DTL_ADD in PROCESS_QUEUE_RECORD.

If the message type is a header delete (HDR_DEL)

- I_hdr_published should never be 'N' (not published), because in that case the HDR_DEL message would never be added to the queue based on ADDTOQ processing.
- Call BUILD_HEADER_OBJECT to build the REF Oracle Object to publish to the RIB.
- Delete both PARTNER_MFQUEUE by calling internal function DELETE_QUEUE_REC and PARTNER_PUB_INFO for the business object.

Otherwise, if check I_hdr_published is either 'N' (not published) or 'I' (in progress)

- If I_hdr_published is 'N', the message type will be a header create (HDR_ADD). If I_hdr_published is 'I', change the message type from a header create (HDR_ADD) to a detail add (DTL_ADD), because this is the situation where some of the details are published but not all due to MAX_DETAILS_TO_PUBLISH. So a header create message for the current business object should have already been published.
- Call MAKE_CREATE to build the DESC Oracle Object to publish to the RIB. This will also take care of any PARTNER_MFQUEUE deletes, updating PARTNER_PUB_INFO.PUBLISHED to 'Y' or 'I', and bulk updating addr.publish_ind to 'Y' for those detail rows that have been published.

Otherwise,

If the record from PARTNER_QUEUE table is a header update (HDR_UPD)

- Call BUILD_HEADER_OBJECT to build the DESC Oracle Object to publish to the RIB.
- Delete the record from the PARTNER_MFQUEUE table.

If the record from PARTNER_QUEUE table is a detail add or update (DTL_ADD, DTL_UPD)

- Call BUILD_DETAIL_CHANGE_OBJECTS to build the DESC Oracle Object to publish to the RIB. This will also take care of any PARTNER_MFQUEUE deletes and updates of publish_ind on ADDR.

If the record from PARTNER_QUEUE table is a detail delete (DTL_DEL)

- Call BUILD_DETAIL_DELETE_OBJECTS to build the REF Oracle Object to publish to the RIB. This will also take care of any PARTNER_MFQUEUE deletes.

Function Level Description – MAKE_CREATE (local)

This private function is used to create the Oracle Object for the initial publication of a business transaction. I_business_object contains the partner header key values (partner type and partner_id). I_rowid is the rowid of the partner_mfqueue row fetched from GETNXT.

- Call BUILD_HEADER_OBJECT to get a header level Oracle Object.
- Call BUILD_DETAIL_OBJECTS to get a table of detail level Oracle objects and a table of PARTNER_MFQUEUE rowids to delete with and a table of ADDR rowids to update publish_ind with.
- Update PARTNER_PUB_INFO.published to 'Y' or 'I' depending on if all details are published.
- Delete records from the PARTNER_MFQUEUE for all rowids returned by BUILD_DETAIL_OBJECTS. Deletes are done by rowids instead of business transaction keys to ensure that noting is deleted off the queue that has not been published.
- If the entire business transaction was added to the Oracle Object, also delete the PARTNER_MFQUEUE record that was picked up by GETNXT. If the entire business transaction was **not** published, the system needs to leave some data on the PARTNER_MFQUEUE to ensure that the rest of it is picked up by the next call to GETNXT.
- Update ADDR.publish_ind to 'Y' by addr rowids returned from BUILD_DETAIL_OBJECTS.
- The header and detail level Oracle Objects are combined and returned.

Function Level Description – BUILD_HEADER_OBJECT (local)

```
Function: BUILD_HEADER_OBJECT
               (O_error_msg                 OUT            VARCHAR2,
                O_rib_partnerdesc_rec       IN OUT NOCOPY  RIB_PARTNERDESC_REC,
                I_business_obj              IN             PARTNER_KEY_REC)
```

This private function accepts partner header key values (partner type and partner ID), builds and returns a header level DESC Oracle Object.

Function Level Description – BUILD_HEADER_OBJECT (local)

This overloaded private function accepts partner header key values (partner type and partner ID), builds and returns a header level REF Oracle Object.

Function Level Description – BUILD_DETAIL_OBJECTS (local)

The private function is responsible for building detail level DESC Oracle Objects. It builds as many detail Oracle Objects as it can given the passed in message type and business object keys (partner type and partner ID).

Call API_LIBRARY.GET_RIB_SETTINGS to get the MAX_DETAILS_TO_PUBLISH for the partner family.

If the function is being called from MAKE_CREATE (I_message_type would be NULL):

Select any unpublished ADDR detail records for the business transaction (based on publish_ind on ADDR). Create Oracle Objects for details that are selected by calling BUILD_SINGLE_DETAIL.

- Ensure that no more than MAX_DETAILS_TO_PUBLISH records are put into Oracle Objects.
- Return a table of partner_mfqueue rowids for each message that is placed into the Oracle Object.
- Return a table of addr rowids for each detail that is placed into the Oracle Object.

If the function is not being called from MAKE_CREATE (I_message type will **not** be NULL):

Select any details on the PARTNER_MFQUEUE that are for the same business object and for the same message type. Create Oracle Objects for details that are selected by calling BUILD_SINGLE_DETAIL.

- Ensure that no more than MAX_DETAILS_TO_PUBLISH records are put into Oracle Objects.
- If the message type is a detail create (DTL_ADD), as the detail info is placed into the Oracle Object, ensure that the corresponding addr rowid is added to addr rowids table for return. These rowids will be used to update ADDR.publish_ind to ‘Y’.
- Return a table of partner_mfqueue rowids for each message that is placed into the Oracle Object.

A concern here is making sure that the system does not delete information from the queue table that has not been published. For this reason, the system performs deletes by ROWID. The system also tries to get all the data in the same cursor to ensure that the message published matches the deletes performed from the PARTNER_MFQUEUE table regardless of trigger execution during GETNXT calls.

Function Level Description – BUILD_SINGLE_DETAIL (local)

This private function takes in an address record and builds a detail level Oracle Object. Also find out if the address is the primary address of the primary address type and set the DESC Oracle Object accordingly.

Function Level Description – BUILD_DETAIL_CHANGE_OBJECTS (local)

This private function builds a DESC Oracle Object to publish to the RIB for detail create and detail update messages (DTL_ADD, DTL_UPD). I_business_obj contains the header level key values (partner type and partner ID).

- Call BUILD_HEADER_OBJECT to build the header level DESC Oracle Object.
- Call BUILD_DETAIL_OBJECTS to build the detail level DESC Oracle Objects.
- Bulk update addr.publish_ind to 'Y' for the addr rowids returned from BUILD_DETAIL_OBJECTS.
- Bulk delete from partner_mfqueue for the partner_mfqueue rowids returned from BUILD_DETAIL_OBJECTS.

Function Level Description – BUILD_DETAIL_DELETE_OBJECTS (local)

This private function builds a REF Oracle Object to publish to the RIB for detail delete messages (DTL_DEL). I_business_obj contains the header level key values (partner type and partner ID).

- Call API_LIBRARY.GET_RIB_SETTINGS to get the MAX_DETAILS_TO_PUBLISH for the partner family.
- Call BUILD_HEADER_OBJECT to build the REF Oracle Object to publish to the RIB.
- Perform a cursor for loop on PARTNER_MFQUEUE and build as many detail REF Oracle Objects as possible without exceeding the MAX_DETAILS_TO_PUBLISH.
- For each detail, also find out if the address is a primary address of the primary address type, and set the REF Oracle Objects accordingly.
- Bulk delete from PARTNER_MFQUEUE for the PARTNER_MFQUEUE rowids queried.

Function Level Description – LOCK_THE_BLOCK (local)

This private function locks all queue records for the current business object (partner type and partner ID). This is to ensure that GETNXT and PUB_RETRY do not wait on any business processes that currently have the queue table locked and have not committed. This can occur because ADDTOQ, which is called from the triggers, deletes from the queue table for DTL_UPD, DTL_DEL, and HDR_DEL messages.

Function Level Description – HANDLE_ERRORS (local)

This private procedure is called from GETNXT and PUB_RETRY when an exception is raised. I_seq_no is the sequence number of the driving PARTNER_MFQUEUE record. I_function_keys contains detail level key values (partner_type, partner_id, addr_key).

If the error is a non-fatal error, HANDLE_ERRORS passes the sequence number of the driving PARTNER_MFQUEUE record back to the RIB in the ROUTING_INFO. It sends back a status of 'H'ospital to the RIB as well. It then updates the status of the queue record to 'H'ospital, so that it will not get picked up again by the driving cursor in GETNXT.

If the error is a fatal error, a status of 'E'rror is returned to the RIB. The error is considered non-fatal if no DML has occurred yet. Whenever DML has occurred, then the global variable LP_error_status is flipped from 'H'ospital to 'E'rror.

Function Level Description – DELETE_QUEUE_REC (local)

This private function will delete the records from PARTNER_MFQUEUE table for the sequence no passed in as input parameter.

Trigger Impact

Trigger Name: EC_TABLE_PRT_AIUDR.TRG (new)

Trigger File Name: ec_table_prt_aiudr.trg (new)

Table: PARTNER

This is the trigger on the PARTNER table that will capture Inserts, Updates, and Deletes.

Inserts

- Send the header level partner info to the ADDTOQ procedure in the MFM with the message type RMSMFM_PARTNER.HDR_ADD.

Updates

- Send the header level partner info to the ADDTOQ procedure in the MFM with the message type RMSMFM_PARTNER.HDR_UPD.

Deletes

- Send the header level info to the ADDTOQ procedure in the MFM with the message type RMSMFM_PARTNER.HDR_DEL.

In all these cases, build the function keys for ADDTOQ with partner type and partner ID.

Trigger Name: EC_TABLE_ADR_AIUDR.TRG (mod)

Trigger File Name: ec_table_adr_aiudr.trg (mod)

Table: ADDR

This is the trigger on the ADDR table that will capture Inserts, Updates, and Deletes of module type ‘PTNR’. (Note: It also handles module types supplier, store and warehouse.)

Inserts

- Send the detail level addr info to the ADDTOQ procedure in the MFM with the message type RMSMFM_PARTNER.DTL_ADD.

Updates

- Send the detail level addr info to the ADDTOQ procedure in the MFM with the message type RMSMFM_PARTNER.DTL_UPD.

Deletes

- Send the detail level info to the ADDTOQ procedure in the MFM with the message type RMSMFM_PARTNER.DTL_DEL.

In all these cases, build the function keys for ADDTOQ with KEY_VALUE_1 and KEY_VALUE_2 and ADDR_KEY, which represent partner_type, partner_id and addr_key respectively.

DTD

Here are the filenames that correspond with each message type. Please consult the RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
PartnerCre	Partner Create Message	PartnerDesc.dtd
PartnerMod	Partner Modify Message	PartnerDesc.dtd
PartnerDel	Partner Delete Message	PartnerRef.dtd
PartnerDtlCre	Partner Detail Create Message	PartnerDtlDesc.dtd
PartnerDtlMod	Partner Detail Modify Message	PartnerDtlDesc.dtd
PartnerDtlDel	Partner Detail Delete Message	PartnerDtlRef.dtd

Table Impact

TABLE	SELECT	INSERT	UPDATE	DELETE
PARTNER_PUB_INFO	Yes	Yes	Yes	Yes
PARTNER_MFQUEUE	Yes	Yes	Yes	Yes
PARTNER	Yes	No	No	No
ADDR	Yes	No	Yes	No
ADD_TYPE_MODULE	Yes	No	No	No
RIB_SETTINGS	Yes	No	No	No

Design Assumptions

- It is not possible for a detail trigger to accurately know the status of a header table. In order for the detail triggers to accurately know when to add a message to the queue, RMS should **not** allow approval of a business object while detail modifications are being made.
- It is not possible for a header trigger or a detail trigger to know the status of anything modified by GETNXT. If a header trigger or detail trigger is trying to delete queue records that GETNXT currently has locked, it will have to wait until GETNXT is finished and removes the lock. It is assumed that this time will be fairly short (at most 2-3 seconds). It is also assumed that this will occur rarely because it involves updating/deleting detail records on a business object that has already been approved. This also has to occur at the same time GETNXT is processing the current business object.
- Push off all DML statements as late as possible. Once DML statements have taken place, any error becomes a fatal error rather than a hospital error.

Performance Considerations

When passing arrays between functions, make sure to use the NOCOPY clause for the array parameter.

This is a low volume API.

Receiver Unit Adjustment

Business Overview

When mistakes are made during the receiving process at the store or warehouse, receiver unit adjustments (RUAs) are made to correct the mistake. RMS publishes messages about receiver unit adjustments to the Oracle Retail Integration Bus (RIB).

When RUAs are initiated through Oracle Retail Invoice Matching (ReIM) or created through RMS forms, a message is published to integration subsystems and onto a Warehouse Management System such as RWMS. Because these systems only have access to the original receipt, the message communicates the original receipt number and not the child receipt number.

When receipt adjustments are made in RMS either through ReIM or the RMS Receiver Unit Adjustment form, it is necessary to communicate the new inventory positions to integration subsystems such as RWMS that track inventory positions. A message is published to the RIB to accomplish this task.

Functionality Checklist

Description	RMS	RIB
RMS must publish RcvUnitAdj information		
Create new Publisher	X	X

Form Impact

None.

Business Object Records

Create the following record types in the RMSMFM_RCVUNITADJ package specification:

```
TYPE rcvunitadj_key_rec IS RECORD(
                            ORDER_NO            SHIPMENT.ORDER_NO%TYPE,
                            ASN                 SHIPMENT.ASN%TYPE,
                            LOCATION            SHIPMENT.TO_LOC%TYPE,
                            LOC_TYPE            SHIPMENT.TO_LOC_TYPE%TYPE,
                            ITEM                SHIPSKU.ITEM%TYPE,
                            CARTON              SHIPSKU.CARTON%TYPE,
                            ADJ_QTY             RUA_MFQUEUE.ADJ_QTY%TYPE);
```

Package Impact

Business Object ID

The location ID will be used as the business object ID. For each location, all receiver unit adjustments (that have not already been published) are rolled up to the item level (each location may have one or more items for which a RUA has been applied) and will be published.

Package Name: RMSMFM_RCVUNITADJ

Spec File Name: rmsmfm_rcvunitadjs.pls

Body File Name: rmsmfm_rcvunitadjb.pls

Package Specification – Global Variables

```
FAMILY            CONSTANT   RIB_SETTINGS.FAMILY%TYPE      'rcvunitadj';
RCVUNITADJ_ADD    CONSTANT   VARCHAR2(15)                  'rcvunitadjcre';
```

Function Level Description – ADDTOQ

```
Function: ADDTOQ(  O_error_msg       IN OUT      VARCHAR2,
                   I_message_type    IN          VARCHAR2,
                   I_business_obj    IN          RCVUNITADJ_KEY_REC)
```

If multi-threading is being used, call API_LIBRARY.RIB_SETTINGS to get the number of threads used for the publisher. Using the number of threads, and the business object ID, calculate the thread value.

Insert a record into the RCVUNITADJ_MFQUEUE.

Function Level Description – GETNXT

```
Procedure: GETNXT GETNXT( O_status_code     OUT     VARCHAR2,
                          O_error_msg       OUT     VARCHAR2,
                          O_message_type    OUT     VARCHAR2,
                          O_message         OUT     RIB_OBJECT,
                          O_bus_obj_id      OUT     RIB_BUSOBJID_TBL,
                          O_routing_info    OUT     RIB_ROUTINGINFO_TBL,
                          I_num_threads     IN      NUMBER DEFAULT 1,
                          I_thread_val      IN      NUMBER DEFAULT 1)
```

The RIB calls GETNXT to get messages. The driving cursor will query for unpublished records on the RCVUNITADJ_MFQUEUE table (PUB_STATUS = 'U').

GETNXT should check for records on the queue with a status of 'H'ospital for the current business object, GETNXT should raise an exception to send the current message to the Hospital.

The information from the RCVUNITADJ_MFQUEUE table is passed to PROCESS_QUEUE_RECORD. PROCESS_QUEUE_RECORD will build the Oracle Object message to pass back to the RIB. If PROCESS_QUEUE_RECORD does not run successfully, GETNXT should raise an exception.

If any exception is raised in GETNXT, including the exception raised by an unsuccessful call to PROCESS_QUEUE_RECORD, HANDLE_ERRORS should be called.

Function Level Description – PUB_RETRY

```
Procedure: PUB_RETRY( O_status_code          OUT           VARCHAR2,
                      O_error_msg            OUT           VARCHAR2,
                      O_message_type         IN  OUT       VARCHAR2,
                      O_message              OUT           RIB_OBJECT,
                      O_bus_obj_id           IN  OUT       RIB_BUSOBJID_TBL,
                      O_routing_info         IN  OUT       RIB_ROUTINGINFO_TBL,
                      I_ref_object           IN            RIB_OBJECT)
```

Same as GETNXT except:

The record on RCVUNITADJ_MFQUEUE must match the passed in sequence number (contained in the ROUTING_INFO).

Function Level Description – PROCESS_QUEUE_RECORD (local)

This function controls the building of Oracle Objects given the business transaction's key values and a message type. It contains all of the shared processing between GETNXT and PUB_RETRY.

The function will first call MAKE_CREATE to build the appropriate oracle object. It then calls the DELETE_QUEUE_REC to delete the RUA_MFQUEUE for the passed-in rowid.

Function Level Description – MAKE_CREATE (local)

This function is used to create the Oracle Object for the initial publication of a business transaction.

- Call BUILD_HEADER_OBJECT to get a header level Oracle Object plus any extra functional holders.
- Call BUILD_DETAIL_OBJECTS to get a table of detail level Oracle objects and add the detail level Oracle Objects to the header object.

Function Level Description – BUILD_HEADER_OBJECT (local)

Accepts header key values, performs necessary lookups, builds and returns a header level Oracle Object.

This function also builds the routing information object using the location.

Function Level Description – BUILD_DETAIL_OBJECTS (local)

The function is responsible for the Oracle Object used for a DESC message (inserts and updates.) It adds as many mfqueue records to the message as it can given the passed in message type and business object keys.

- Call BUILD_SINGLE_DETAIL passing in the I_business_obj record.
- Ensure that ROUTING_INFO is constructed if routing information is stored at the detail level in the business transaction.

Function Level Description – BUILD_SINGLE_DETAIL (local)

Accept inputs and build a detail level Oracle Object. If the adjustment quantity is negative, the from disposition should be 'ATS' and the to disposition should be NULL. If the adjustment quantity is positive, the to disposition should be NULL and the from disposition should be 'ATS'.

Function Level Description – LOCK_THE_BLOCK (local)

This function locks all queue records for the current business object. This is to ensure that GETNXT does not wait on any business processes that currently have the queue table locked and have not committed. This can occur because ADDTOQ, which is called from the triggers, deletes from the queue table for DTL_UPD, DTL_DEL, and HDR_DEL messages.

Function Level Description – HANDLE_ERRORS (local)

HANDLE_ERRORS is called from GETNXT and PUB_RETRY when an exception is raised.

If the error is a non-fatal error, GETNXT passes the sequence number of the driving RCVUNITADJ_MFQUEUE record back to the RIB in the ROUTING_INFO. It sends back a status of 'H'ospital to the RIB as well. It then updates the status of the queue record to 'H'ospital, so that it will not get picked up again by the driving cursor in GETNXT.

If the error is a fatal error, a status of 'E'rror is returned to the RIB.

The error is considered non-fatal if no DML has occurred yet. Whenever DML has occurred, then the global variable LP_error_status is flipped from 'H'ospital to 'E'rror.

Function Level Description – DELETE_QUEUE_REC (local)

This private function will delete the records from rcvunitadj_mfqueue table for the rowid passed in as input parameter.

Trigger Impact

Trigger Name: EC_TABLE_RUA_AIR.TRG

Trigger File Name: ec_table_rua_air.trg

Table: RAU_RIB_INTERFACE

Create a trigger on the RAU_RIB_INTERFACE table to capture Inserts.

Inserts

- Send the appropriate column values to the ADDTOQ procedure in the MFM with the message type RMSMFM_RCVUNITADJ.RCVUNITADJ_ADD.

DTD

Here are the filenames that correspond with each message type. Please consult RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
RcvUnitAdjCre	Receiver Unit Adjustment Create Message	RcvUnitAdjDesc.dtd

Table Impact

TABLE	SELECT	INSERT	UPDATE	DELETE
RUA_MFQUEUE	Yes	Yes	Yes	Yes

Design Assumptions

- It is not possible for a detail trigger to accurately know the status of a header table.
- In order for the detail triggers to accurately know when to add a message to the queue, RMS should not allow approval of a business object while detail modifications are being made.
- It is not possible for a header trigger or a detail trigger to know the status of anything modified by GETNXT. If a header trigger or detail trigger is trying to delete queue records that GETNXT currently has locked, it will have to wait until GETNXT is finished and removes the lock. It is assumed that this time will be fairly short (at most 2-3 seconds.) It is also assumed that this will occur rarely because it involves updating/deleting detail records on a business object that has already been approved. This also has to occur at the same time GETNXT is processing the current business object.
- Push off all DML statements as late as possible. Once DML statements have taken place, any error becomes a fatal error rather than a hospital error.

RTV Request

Business Overview

A return to vendor (RTV) order is used to send merchandise back to the supplier. The RTV message is published by RMS to the store. For an RTV, the initial transfer of stock to the store is a distinctly different step from the RTV itself. Once the transferred stock arrives at the store, the user then creates the RTV. RTVs are created by the following:

1. Adding one supplier
2. Selecting the sending locations
3. Adding the items, either individually or through the use of item lists

In order to return items to a vendor from multiple stores as part of one operation, the items must go through a single warehouse. The transfer of items from several different stores to one warehouse is referred to as a mass return transfer (MRT). The items are subsequently returned to the vendor from the warehouse.

Return to vendor requests created in RMS should be published to the RIB to allow the integration subsystem application to have visibility to the corporately created RTV. Consequently, when the integration subsystem application ships the RTV, it must communicate the original RTV order number back to RMS so that RMS can correctly update the original RTV record.

Functionality Checklist

Description	RMS	RIB
RMS must publish RTV information		
Publish RTV information from RMS to the RIB.	X	X

Form Impact

None.

Business Object Records

None.

Package Impact

Business Object ID

No change.

Package Name: RMSMFM_RTVREQ

Spec File Name: rmsmfm_rtvreqs.pls

Body File Name: rmsmfm_rtvreqb.pls

Package Specification – Global Variables

No change.

Function Level Description – ADDTOQ

```
Function:  ADDTOQ
               (O_error_msg          IN OUT  VARCHAR2,
                I_message_type       IN      VARCHAR2,
                I_rtv_order_no       IN      RTV_HEAD.RTV_ORDER_NO%TYPE,
                I_status             IN      RTV_HEAD.STATUS_IND%TYPE,
                I_rtv_seq_no         IN      RTV_DETAIL.SEQ_NO%TYPE,
                I_item               IN      RTV_DETAIL.ITEM%TYPE,
                I_publish_ind        IN      RTV_DETAIL.PUBLISH_IND%TYPE)
```

There are some tasks relating to streamlining the queue clean up process that need to occur in ADDTOQ. The goal is to have at most one record on the queue for business transactions up until their initial publication.

- For header level insert messages (HDR_ADD), insert a record in the RTVREQ_PUB_INFO table. The published flag should be set to 'N'. The correct thread for the business transaction should be calculated and written. Call API_LIBRARY.RIB_SETTINGS to get the number of threads used for the publisher. Using the number of threads, and the business object id, calculate the thread value.
- For all records except header level inserts (HDR_ADD), the thread_no, initial_approval_ind, and shipped_ind should be queried from the RTVREQ_PUB_INFO table.
- If the business transaction has not been approved (initial_approval_ind = 'N') or it has already been shipped (shipped_ind = 'Y') and the triggering message is one of DTL_ADD, DTL_UPD, DTL_DEL, HDR_DEL, no processing should take place and the function should exit.
- For detail level messages deletes (DTL_DEL), the system only needs one (the most recent) record per detail in the RTVREQ_MFQUEUE. Delete any previous records that exist on the RTVREQ_MFQUEUE for the record that has been passed. If the publish_ind is 'N', do not add the DTL_DEL message to the queue.
- For detail level message deletes (DTL_UPD), the system only needs one DTL_UPD (the most recent) record per detail in the RTVREQ_MFQUEUE. Delete any previous DTL_UPD records that exist on the RTVREQ_MFQUEUE for the record that has been passed. The system does not want to delete any detail inserts that exist on the queue for the detail. The system needs to ensure subscribers are not passed a detail modification message for a detail that they do not yet have.
- For header level delete messages (HDR_DEL), delete every record in the queue for the business transaction.
- For header level update message (HDR_UPD), update the RTVREQ_PUB_INFO.INITIAL_APPROVAL_IND to 'Y' if the business transaction is in approved status (status of '10').
- For header level update message (HDR_UPD), update the RTVREQ_PUB_INFO.SHIPPED_IND to 'Y' if the business transaction is in shipped status (status of '15').
- For all records except header level inserts (HDR_ADD), insert a record into the RTVREQ_MFQUEUE.

Function Level Description – GETNXT

```
Procedure: GETNXT
              (O_status_code        OUT    VARCHAR2,
               O_error_msg          OUT    VARCHAR2,
               O_message_type       OUT    VARCHAR2,
               O_message            OUT    RIB_OBJECT,
               O_bus_obj_id         OUT    RIB_BUSOBJID_TBL,
               O_routing_info       OUT    RIB_ROUTINGINFO_TBL,
               I_num_threads        IN     NUMBER DEFAULT 1,
               I_thread_val         IN     NUMBER DEFAULT 1)
```

Make sure to initialize LP_error_status to API_CODES.HOSPITAL at the beginning of GETNXT.

The RIB calls GETNXT to get messages. It performs a cursor loop on the unpublished records on the RTVREQ_MFQUEUE table (PUB_STATUS = 'U'). It should only need to execute one loop iteration in most cases. For each record retrieved, GETNXT gets the following:

1. A lock of the queue table for the current business object. The lock is obtained by calling the function LOCK_THE_BLOCK. If there are any records on the queue for the current business object that are already locked, the current message is skipped.
2. The published indicator from the RTVREQ_PUB_INFO table.
3. A check for records on the queue with a status of 'H'ospital. If there are any such records for the current business object, GETNXT raises an exception to send the current message to the Hospital.

The loop will need to execute more than one iteration for the following cases:

1. When a header delete message exists on the queue for a business object that has not been initially published. In this case, simply remove the header delete message from the queue and loop again.
2. The queue is locked for the current business object.

The information from the RTVREQ_MFQUEUE and RTVREQ_PUB_INFO table is passed to PROCESS_QUEUE_RECORD. PROCESS_QUEUE_RECORD will build the Oracle Object message to pass back to the RIB. If PROCESS_QUEUE_RECORD does not run successfully, GETNXT raises an exception.

If any exception is raised in GETNXT, including the exception raised by an unsuccessful call to PROCESS_QUEUE_RECORD, HANDLE_ERRORS is called.

Function Level Description – PUB_RETRY

Procedure: PUB_RETRY

Same as GETNXT except:

The record on RTVREQ_MFQUEUE must match the passed in sequence number (contained in the ROUTING_INFO).

Function Level Description – PROCESS_QUEUE_RECORD (local)

This function controls the building of Oracle Objects given the business transaction's key values and a message type. It contains all of the shared processing between GETNXT and PUB_RETRY.

Check to see if the business object is being published for the first time. If the published_ind on the PUB_INFO table is 'N' or 'I', the business object is being published for the first time. If so, call MAKE_CREATE.

Otherwise,

If the record from RTVREQ_MFQUEUE table is a header update (HDR_UPD)

- Call BUILD_HEADER_OBJECT to build the Oracle Object to publish to the RIB. This will also populate the ROUTING_INFO.
- Update RTVREQ_PUB_INFO with updated new header information
- Delete the record from the RTVREQ_MFQUEUE table.

If the record from RTVREQ_MFQUEUE table is a detail add or update (DTL_ADD, DTL_UPD)

- Call BUILD_HEADER_OBJECT to build the header portion of the Oracle Object to publish to the RIB. This will also populate the ROUTING_INFO.
- Call BUILD_DETAIL_CHANGE_OBJECTS to build the detail portion of the Oracle Object. This will also take care of any RTVREQ_MFQUEUE deletes.

If the record from RTVREQ_MFQUEUE table is a detail delete (DTL_DEL)

- Call BUILD_HEADER_OBJECT to build the header portion of the Oracle Object to publish to the RIB. This will also populate the ROUTING_INFO.
- Call BUILD_DETAIL_DELETE_OBJECTS to build the detail portion of the Oracle Object. This will also take care of any RTVREQ_MFQUEUE deletes.

Function Level Description – MAKE_CREATE (local)

This function is used to create the Oracle Object for the initial publication of a business transaction.

- Call BUILD_HEADER_OBJECT to build the Oracle Object to publish to the RIB. This will also populate the ROUTING_INFO.
- Call BUILD_DETAIL_OBJECTS to get a table of detail level Oracle objects and a table of RTVREQ_MFQUEUE rowids to delete.
- Delete records from the RTVREQ_MFQUEUE for all rowids returned by BUILD_DETAIL_OBJECTS. Deletes are done by rowids instead of business transaction keys to ensure that noting is deleted off the queue that has not been published.
- If the entire business transaction was added to the Oracle Object, also delete the RTVREQ_MFQUEUE record that was picked up by GETNXT. If the entire business transaction was not published the system needs to leave something on the RTVREQ_MFQUEUE to ensure that the rest of it is picked up by the next call to GETNXT.
- The header and detail level Oracle Objects are combined and returned.

Function Level Description – BUILD_HEADER_OBJECT (local)

Take all necessary data from RTV_HEAD table and put it into a RIB_RTVREQDESC_REC and RIB_RTVREQREF_REC object.

Put the location into the ROUTING_INFO.

Function Level Description – BUILD_DETAIL_CHANGE_OBJECTS (local)

Call BUILD_DETAIL_OBJECTS.

BUILD_DETAIL_OBJECTS creates a table of RTVREQ_MFQUEUE ROWIDs to delete. Delete these records.

BUILD_DETAIL_OBJECTS creates a table of RTV_DETAIL ROWIDs to update. Update the PUBLISH_IND to Y for these records.

Make sure to set LP_error_status to API_CODES.UNHANDLED_ERROR before any DML statements.

Function Level Description – BUILD_DETAIL_OBJECTS (local)

The function is responsible for building detail level Oracle Objects. It builds as many detail Oracle Object as it can given the passed in message type and business object keys.

If the function is being called from MAKE_CREATE:

Select any unpublished detail records from the business transaction (RTV_DETAIL.PUBLISH_IND will be 'N'). Create Oracle Objects for details that are selected by calling BUILD_SINGLE_DETAIL.

- Ensure that the PUBLISH_IND gets set to Y for each RTV_DETAIL record placed into the Oracle Objects. A table of ROWIDs to update will be created in BUILD_DETAIL_OBJECTS. The actual update statement will occur in BUILD_DETAIL_CHANGE_OBJECTS using this table of ROWIDS.
- Ensure that RTVREQ_MFQUEUE is deleted from as needed. If there is more than one RTVREQ_MFQUEUE record for a detail level record, make sure they all get deleted. The system only cares about current state, not every change. A table of ROWIDs to delete will be created in BUILD_DETAIL_OBJECTS. The actual delete statement will occur in BUILD_DETAIL_CHANGE_OBJECTS using this table of ROWIDS.
- Ensure that no more than MAX_DETAILS_TO_PUBLISH records are put into Oracle Objects.
- Ensure that the detail records being added to the object have not already been published. This can happen if GETNXT was previously called for the current business object, and the MAX_DETAILS_TO_PUBLISH limit had been reached. The system ensures these details do not get added again by looking at each detail's PUBLISH_IND.

If the function is not being called from MAKE_CREATE:

- Select any records on the RTVREQ_MFQUEUE that are for the same business object ID. Fetch the records in order of seq_no on the MFQUEUE table.
- Ensure that RTVREQ_MFQUEUE is deleted from as needed. A table of ROWIDs to delete will be created in BUILD_DETAIL_OBJECTS. The actual delete statement will occur in BUILD_DETAIL_CHANGE_OBJECTS using this table of ROWIDS.

- If the message type is a detail create (DTL_ADD), ensure that the PUBLISH_IND gets set to Y for each RTV_DETAIL record placed into the Oracle Objects. A table of ROWIDs to update will be created in BUILD_DETAIL_OBJECTS. The actual update statement will occur in BUILD_DETAIL_CHANGE_OBJECTS using this table of ROWIDS.

A concern here is making sure that the system does not delete information from the queue table that has not been published. For this reason, the system performs deletes by ROWID. The system also attempts to get everything in the same cursor to ensure that the message published matches the deletes performed from the RTVREQ_MFQUEUE table regardless of trigger execution during GETNXT calls.

Function Level Description – BUILD_DETAIL_DELETE_OBJECTS (local)

This function works the same way as BUILD_DETAIL_OBJECTS, except for the fact that a REF object is being created instead of a DESC object.

Function Level Description – BUILD_SINGLE_DETAIL (local)

This function puts the inputted information in a RIB_RTVREQDTL_TBL object.

Function Level Description – LOCK_THE_BLOCK (local)

This function locks all queue records for the current business object. This is to ensure that GETNXT does not wait on any business processes that currently have the queue table locked and have not committed. This can occur because ADDTOQ, which is called from the triggers, deletes from the queue table for DTL_UPD, DTL_DEL, and HDR_DEL messages.

Function Level Description – DELETE_QUEUE_REC (local)

This function deletes a record from the RTVREQ_MFQUEUE table, using the passed in sequence number.

Function Level Description – HANDLE_ERRORS (local)

HANDLE_ERRORS is called from GETNXT and PUB_RETRY when an exception is raised.

If the error is a non-fatal error, GETNXT passes the sequence number of the driving ITEMLOC_MFQUEUE record back to the RIB in the ROUTING_INFO. It sends back a status of ‘H’ospital to the RIB as well. It then updates the status of the queue record to ‘H’ospital, so that it will not get picked up again by the driving cursor in GETNXT.

If the error is a fatal error, a status of ‘E’rror is returned to the RIB.

The error is considered non-fatal if no DML has occurred yet. Whenever DML has occurred, then the global variable LP_error_status is flipped from ‘H’ospital to ‘E’rror.

Trigger Impact

Trigger Name: EC_TABLE_RHD_AIUDR.TRG

Trigger File Name: ec_table_rhd_aiudr.trg

Table: RTV_HEAD

Create a trigger on the RTV_HEAD table to capture Inserts, Updates, and Deletes.

Inserts

- Send the appropriate column values to the ADDTOQ procedure in the MFM with the message type RMSMFM_RTVREQ.HDR_ADD.

Updates

- Send the appropriate column values to the ADDTOQ procedure in the MFM with the message type RMSMFM_RTVREQ.HDR_UPD.

Deletes

- Send the appropriate column values to the ADDTOQ procedure in the MFM with the message type RMSMFM_RTVREQ.HDR_DEL.

Trigger Name: EC_TABLE_RDT_AIUDR.TRG

Trigger File Name: ec_table_rdt_aiudr.trg

Table: RTV_DETAIL

Create a trigger on the RTV_DETAIL table to capture Inserts, Updates, and Deletes.

Inserts

- Send the appropriate column values to the ADDTOQ procedure in the MFM with the message type RMSMFM_RTVREQ.DTL_ADD.

Updates

- Send the appropriate column values to the ADDTOQ procedure in the MFM with the message type RMSMFM_RTVREQ.DTL_UPD.

Deletes

- Send the appropriate column values to the ADDTOQ procedure in the MFM with the message type RMSMFM_RTVREQ.DTL_DEL.

DTD

Here are the filenames that correspond with each message type. Please consult the RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
RtvReqCre	RTV Request Create Message	RTVReqDesc.dtd
RtvReqMod	RTV Request Modify Message	RTVReqDesc.dtd
RtvReqDel	RTV Request Delete Message	RTVReqRef.dtd
RtvReqDtlCre	RTV Request Detail Create Message	RTVReqDesc.dtd
RtvReqDtlMod	RTV Request Detail Modify Message	RTVReqDesc.dtd
RtvReqDtlDel	RTV Request Detail Delete Message	RTVReqRef.dtd

Table Impact

TABLE	SELECT	INSERT	UPDATE	DELETE
RTVREQ_MFQUEUE	Yes	Yes	Yes	Yes
RTVREQ_PUB_INFO	Yes	Yes	Yes	Yes
RTV_HEAD	Yes	No	No	No
RTV_DETAIL	Yes	No	No	No

Design Assumptions

- It is not possible for a detail trigger to accurately know the status of a header table.
- In order for the detail triggers to accurately know when to add a message to the queue, RMS should not allow approval of a business object while detail modifications are being made.
- It is not possible for a header trigger or a detail trigger to know the status of anything modified by GETNXT. If a header trigger or detail trigger is trying to delete queue records that GETNXT currently has locked, it will have to wait until GETNXT is finished and remove the lock. It is assumed that this time will be fairly short (at most 2-3 seconds.) It is also assumed that this will occur rarely because it involves updating/deleting detail records on a business object that has already been approved. This also has to occur at the same time GETNXT is processing the current business object.
- Push off all DML statements as late as possible. Once DML statements have taken place, any error becomes a fatal error rather than a hospital error.

Seed Data

Business Overview

Seed data publication to the RIB allows RMS to send code information and differentiator type information to external systems through an integration subsystem.

Some examples of seed data include item types, carriers, shipping methods, and return reasons. Such seed data is usually fairly constant and unchanging.

Seed data is used the first time that a system loads data.

Seed data publication uses CLOB messages.

Code information as well as differentiator types will be published to the RIB so that external systems will have all of the latest information regarding codes and diff types. Previously, all of the codes and diff types were populated at installation time and when codes or diff types were added or modified. External systems would not get these changes.

Seed data publication consists of a message containing code and diff type information from the tables CODE_HEAD, CODE_DETAIL and DIFF_TYPE. One message will be synchronously created and placed in the message queue each time a CODE_HEAD, CODE_DETAIL or DIFF_TYPE record is created, modified, or deleted. When a CODE_HEAD, CODE_DETAIL or DIFF_TYPE record is created or modified, the message will contain a full snapshot of the modified record. When a CODE_HEAD, CODE_DETAIL or DIFF_TYPE record is deleted, the message will contain a partial snapshot of the deleted record. Messages are retrieved from the message queue in the order they were created.

Functionality Checklist

Description	RMS	RIB
RMS must publish code and diff type information		
Create new Publisher	X	X

Form Impact

None.

Business Object Records

N/A

Package Impact

Package Name: RMSMFM_SEEDDATA

Spec File Name: rmsmfm_seeddatas.pls

Body File Name: rmsmfm_seeddatab.pls

Package Specification – Global Variables

```
/*--- message type parameters ---*/
HDR_CRE_TYPE           VARCHAR2(30)   := 'CodeHdrCre';
HDR_MOD_TYPE           VARCHAR2(30)   := 'CodeHdrMod';
HDR_DEL_TYPE           VARCHAR2(30)   := 'CodeHdrDel';
DTL_CRE_TYPE           VARCHAR2(30)   := 'CodeDtlCre';
DTL_MOD_TYPE           VARCHAR2(30)   := 'CodeDtlMod';
DTL_DEL_TYPE           VARCHAR2(30)   := 'CodeDtlDel';
DIFF_TYPE_CRE_TYPE     VARCHAR2(30)   := 'DiffTypeCre';
DIFF_TYPE_MOD_TYPE     VARCHAR2(30)   := 'DiffTypeMod';
DIFF_TYPE_DEL_TYPE     VARCHAR2(30)   := 'DiffTypeDel';

/*--- doc type parameters ---*/
HDR_DESC_MSG           CONSTANT  VARCHAR2(30)       := 'CodeHdrDesc';
HDR_REF_MSG            CONSTANT  VARCHAR2(30)       := 'CodeHdrRef';
DTL_DESC_MSG           CONSTANT  VARCHAR2(30)       := 'CodeDtlDesc';
DTL_REF_MSG CONSTANT   VARCHAR2(30)                 := 'CodeDtlRef';
DIFF_TYPE_DESC_MSG     CONSTANT  VARCHAR2(30)       := 'DiffTypeDesc';
DIFF_TYPE_REF_MSG      CONSTANT  VARCHAR2(30)       := 'DiffTypeRef';
```

Function Level Description – ADDTOQ

```
PROCEDURE: ADDTOQ
               (O_status      OUT           VARCHAR2,
                O_text        OUT           VARCHAR2,
            I_message_type    IN            CODES_MFQUEUE.MESSAGE_TYPE%TYPE,
            I_code_type       IN            CODES_MFQUEUE.CODE_TYPE%TYPE,
            I_message         IN OUT        rib_sxw.SXWHandle)
```

This procedure is called by EC_TABLE_CODEHD_AIUDR, EC_TABLE_CODEDTL_AIUDR and EC_TABLE_DIFF_TYPE_AIUDR. The procedure accepts a message variable that consists of the code or diff information in XML tags, a code type variable (this will be hard coded ‘OOOO’ for diff types) and one of the message types defined in the package specification. It inserts a row into the message family queue CODES_MFQUEUE along with the passed in values and the next sequence number from the message family sequence, and sets the status to unpublished. The procedure will then call API_LIBRARY.WRITE_DOCUMENT_STR which will return a status code of API_CODES.SUCCESS if successful, API_CODES.UNHANDLED_ERROR if not.

Function Level Description – GETNXT

```
PROCEDURE GETNXT
          (O_status_code     OUT          VARCHAR2,
           O_error_msg       OUT          VARCHAR2,
           O_message_type    OUT          CODES_MFQUEUE.MESSAGE_TYPE%TYPE,
           O_message         OUT          nocopy CLOB,
           O_code_type       OUT          CODES_MFQUEUE.CODE_TYPE%TYPE)
```

This publicly exposed procedure is typically called by a RIB publication adaptor. Its parameters are well defined and arranged in a specific order. The message type is the RIB defined short message name.

The error text parameter contains application-generated information, such as the application's sequence number of the message that failed, and the Oracle or other error that occurred when the retrieval failed.

This procedure will call the internal function DO_GETNXT which will actually retrieve the clob from the CODES_MFQUEUE table so that it may be published to the RIB.

Function Level Description – DO_GETNXT (local)

This internal procedure will select the record from the CODES_MFQUEUE table having the lowest sequence number and a pub_status of 'U'. It will return the clob, the message type and code type to the out parameters to be passed back to GETNXT. The procedure will then call the DELETE_QUEUE_REC function to delete the record that is being published.

Function Level Description – DELETE_QUEUE_REC (local)

This procedure will delete the record from the CODES_MFQUEUE table that has the sequence number corresponding to the I_seq_no parameter.

Package Name: CODE_HEAD_XML

Spec File Name: code_head_xmls.pls

Body File Name: code_head_xmlb.pls

Package Specification – Global Variables

None.

Function Level Description – BUILD_MESSAGE

If the I_action_type is 'D' (a record is being deleted), an internal variable holding the doc type should be set to RMSMFM_SEEDDATA.HDR_REF_MSG. The function will then call API_LIBRARY.CREATE_MESSAGE_STR to return a clob that has the appropriate structure for the code head delete message. The function then calls the DELETE_CODE_HEAD function to populate the clob that was created.

If the I_action_type is not 'D' (a record has been added or updated), an internal variable holding the doc type should be set to RMSMFM_SEEDDATA.HDR_DESC_MSG. The function will then call API_LIBRARY.CREATE_MESSAGE_STR to return a clob that has the appropriate structure for the code head desc message. The function then calls the ADD_UPDATE_CODE_HEAD function to populate the clob that was created.

Function Level Description – DELETE_CODE_HEAD

This function will accept a record that holds code_head values. The rib_sxw.addElement function will be called to add the code type from the I_code_head_rec to the clob (or root).

Function Level Description – ADD_UPDATE_CODE_HEAD

This function will accept a record that holds CODE_HEAD values. The rib_sxw.addElement function will be called to add the code type and code type description from the I_code_head_rec to the clob (or root).

Package Name: CODE_DETAIL_XML

Spec File Name: code_detail_xmls.pls

Body File Name: code_detail_xmlb.pls

Package Specification – Global Variables

None.

Function Level Description – BUILD_MESSAGE

If the I_action_type is 'D' (a record is being deleted), an internal variable holding the doc type should be set to RMSMFM_SEEDDATA.DTL_REF_MSG. The function will then call API_LIBRARY.CREATE_MESSAGE_STR to return a clob that has the appropriate structure for the code detail delete message. The function then calls the DELETE_CODE_DETAIL function to populate the clob that was created.

If the I_action_type is not 'D' (a record has been added or updated), an internal variable holding the doc type should be set to RMSMFM_SEEDDATA.DTL_DESC_MSG. The function will then call API_LIBRARY.CREATE_MESSAGE_STR to return a clob that has the appropriate structure for the code detail desc message. The function then calls the ADD_UPDATE_CODE_DETAIL function to populate the clob that was created.

Function Level Description – DELETE_CODE_DETAIL

```
FUNCTION DELETE_CODE_DETAIL
               (O_status             OUT     VARCHAR2,
                O_text               OUT     VARCHAR2,
                I_code_detail_rec    IN      CODE_DETAIL%ROWTYPE,
                Root                 IN      OUT rib_sxw.SXWHandle)
```

This function will accept a record that holds code_detail values The rib_sxw.addElement function will be called to add the code type and code from the I_code_detail_rec to the clob (or root).

Function Level Description – ADD_UPDATE_CODE_DETAIL

This function will accept a record that holds code_detail values. The rib_sxw.addElement function will be called to add the code type, code, code description, required indicator and code sequence from the I_code_detail_rec to the clob (or root).

Package Name: DIFF_TYPE_XML

Spec File Name: diff_type_xmls.pls

Body File Name: diff_type_xmlb.pls

Package Specification – Global Variables

None.

Function Level Description – BUILD_MESSAGE

If the I_action_type is 'D' (a record is being deleted), an internal variable holding the doc type should be set to RMSMFM_SEEDDATA.DIFF_TYPE_REF_MSG. The function will then call API_LIBRARY.CREATE_MESSAGE_STR to return a clob that has the appropriate structure for the diff type delete message. The function then calls the DELETE_DIFF_TYPE function to populate the clob that was created.

If the I_action_type is not 'D' (a record has been added or updated), an internal variable holding the doc type should be set to RMSMFM_SEEDDATA.DIFF_TYPE_DESC_MSG. The function will then call API_LIBRARY.CREATE_MESSAGE_STR to return a clob that has the appropriate structure for the diff type desc message. The function then calls the ADD_UPDATE_DIFF_TYPE function to populate the clob that was created.

Function Level Description – DELETE_DIFF_TYPE

```
FUNCTION DELETE_DIFF_TYPE
               (O_status             OUT     VARCHAR2,
                O_text               OUT     VARCHAR2,
                I_diff_type_rec      IN      DIFF_TYPE%ROWTYPE,
                Root                 IN      OUT rib_sxw.SXWHandle)
```

This function will accept a record that holds diff_type values. The rib_sxw.addElement function will be called to add the diff type from the I_diff_type_rec to the clob (or root).

Function Level Description – ADD_UPDATE_CODE_DETAIL

```
FUNCTION ADD_UPDATE_DIFF_TYPE
               (O_status             OUT           VARCHAR2,
                O_text               OUT           VARCHAR2,
                I_diff_type_rec      IN            DIFF_TYPE%ROWTYPE,
                Root                 IN OUT        rib_sxw.SXWHandle)
```

This function will accept a record that holds diff_type values. The rib_sxw.addElement function will be called to add the diff type and diff type description from the I_diff_type_rec to the clob (or root).

Trigger Impact

Trigger Name: EC_TABLE_CODEHD_AIUDR.TRG

Trigger File Name: ec_table_codehd_aiudr.trg

Table: CODE_HEAD

This trigger will capture inserts/updates/deletes to the CODE_HEAD table and write data into the CODES_MFQUEUE message queue.

Inserts

- Set the message type RMSMFM_SEEDDATA.HDR_CRE_TYPE. Set the action type to 'A'. Populate the code head record with the code type and code type description.

Updates

- Set the message type RMSMFM_SEEDDATA.HDR_MOD_TYPE. Set the action type to 'M'. Make sure that either the code type or code type description has changed. If they have not, then simply return and do not add anything to the CODES_MFQUEUE. If either is different, then populate the code head record with the code type and code type description.

Deletes

- Set the message type RMSMFM_SEEDDATA.HDR_DEL_TYPE. Set the action type to 'D'. Populate the code head record with the code type.

Call CODE_HEAD_XML.BUILD_MESSAGE to build the clob that will be placed on the CODES_MFQUEUE. Finally, call the RMSMFM_SEEDDATA.ADDTOQ function with the message type, code type and clob.

Trigger Name: EC_TABLE_CODEDTL_AIUDR.TRG

Trigger File Name: ec_table_codedtl_aiudr.trg

Table: CODE_DETAIL

This trigger will capture inserts/updates/deletes to the CODE_DETAIL table and write data into the CODES_MFQUEUE message queue.

Inserts

- Set the message type RMSMFM_SEEDDATA.DTL_CRE_TYPE. Set the action type to 'A'. Populate the code detail record with the code type, code, code description, required indicator and code sequence.

Updates

- Set the message type RMSMFM_SEEDDATA.DTL_MOD_TYPE. Set the action type to 'M'. Make sure that the code type, code, code description, required indicator or code sequence have changed. If they have not, then simply return and do not add anything to the CODES_MFQUEUE. If either is different, then populate the code detail record with the code type, code, code description, required indicator and code sequence.

Deletes

- Set the message type RMSMFM_SEEDDATA.DTL_DEL_TYPE. Set the action type to ‘D’. Populate the code detail record with the code type and code.

Call CODE_DETAIL_XML.BUILD_MESSAGE to build the clob that will be placed on the CODES_MFQUEUE. Finally, call the RMSMFM_SEEDDATA.ADDTOQ function with the message type, code type and clob.

Trigger Name: EC_TABLE_DIFF_TYPE_AIUDR.TRG

Trigger File Name: ec_table_diff_type_aiudr.trg

Table: DIFF_TYPE

This trigger will capture inserts/updates/deletes to the DIFF_TYPE table and write data into the CODES_MFQUEUE message queue.

Inserts

- Set the message type RMSMFM_SEEDDATA.DIFF_TYPE_CRE_TYPE. Set the action type to ‘A’. Populate the diff type record with the diff type and diff type description.

Updates

- Set the message type RMSMFM_SEEDDATA.DIFF_TYPE_MOD_TYPE. Set the action type to ‘M’. Make sure that the diff type and diff type description have changed. If they have not, then simply return and do not add anything to the CODES_MFQUEUE. If either is different, then populate the diff type record with the diff type and diff type description.

Deletes

- Set the message type RMSMFM_SEEDDATA.DIFF_TYPE_DEL_TYPE. Set the action type to ‘D’. Populate the diff type record with the diff type.

Call DIFF_TYPE_XML.BUILD_MESSAGE to build the clob that will be placed on the CODES_MFQUEUE. Finally, call the RMSMFM_SEEDDATA.ADDTOQ function with the message type, code type and clob. The code type for diff types should always be ‘OOOO’.

DTD

Here are the filenames that correspond with each message type. Please consult the RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
CodeHdrCre	Code Head Create Message	CodeHdrDesc.dtd
CodeHdrMod	Code Head Modify Message	CodeHdrDesc.dtd
CodeHdrDel	Code Head Delete Message	CodeHdrRef.dtd
CodeDtlCre	Code Detail Create Message	CodeDtlDesc.dtd
CodeDtlMod	Code Detail Modify Message	CodeDtlDesc.dtd
CodeDtlDel	Code Detail Delete Message	CodeDtlRef.dtd
DiffTypeCre	Diff Type Create Message	DiffTypeDesc.dtd
DiffTypeMod	Diff Type Modify Message	DiffTypeDesc.dtd
DiffTypeDel	Diff Type Delete Message	DiffTypeRef.dtd

Table Impact

TABLE	SELECT	INSERT	UPDATE	DELETE
CODES_MFQUEUE	Yes	Yes	No	Yes

Design Assumptions

- One of the primary assumptions in the current approach is that ease of code will outweigh performance considerations. It is hoped that the ‘trickle’ nature of the flow of data will decrease the need to dwell on performance issues and instead allow developers to code in the easiest and most straightforward manner.
- The adaptor is only setup to call stored procedures, not stored functions. Any public program called by the adaptor then needs to be a procedure.

Seed Object

Business Overview

Seed object publication to the RIB allows RMS to send country information as well as currency rates so that external systems will have all of the latest information regarding countries and currency rates.

Seed object publication consists of a message containing country and currency rate information from the tables COUNTRY and CURRENCY_RATES. One message will be synchronously created and placed in the message queue each time a COUNTRY and CURRENCY_RATES record is created, modified or deleted in RMS. When a COUNTRY or CURRENCY_RATES record is created or modified, the message will contain a full snapshot of the modified record. When a COUNTRY record is deleted, the message will contain a partial snapshot of the deleted record. Messages are retrieved from the message queue in the order they were created.

Functionality Checklist

Description	RMS	RIB
RMS must publish code and diff type information		
Create new Publisher	X	X

Form Impact

None.

Business Object Records

N/A

Package Impact

Package Name: RMSMFM_SEEDOBJ

Spec File Name: rmsmfm_seedobjs.pls

Body File Name: rmsmfm_seedobjb.pls

Package Specification – Global Variables

```
/*--- message type parameters ---*/
COUNTRY_ADD    VARCHAR2(30)    := 'countrycre';
COUNTRY_UPD    VARCHAR2(30)    := 'countrymod';
COUNTRY_DEL    VARCHAR2(30)    := 'countrydel';
CURR_ADD       VARCHAR2(30)    := 'curratecre';
CURR_UPD       VARCHAR2(30)    := 'curratemod';
```

Function Level Description – ADDTOQ

```
PROCEDURE: ADDTOQ
          (O_error_message   IN OUT         VARCHAR2,
           I_message_type    IN             SEEDOBJ_MFQUEUE.MESSAGE_TYPE%TYPE,
           I_country_id      IN             SEEDOBJ_MFQUEUE.COUNTRY_ID%TYPE,
           I_currency_code   IN             SEEDOBJ_MFQUEUE.CURRENCY_CODE%TYPE,
           I_country_desc    IN             SEEDOBJ_MFQUEUE.COUNTRY_DESC%TYPE,
           I_effective_date  IN             SEEDOBJ_MFQUEUE.EFFECTIVE_DATE%TYPE,
           I_exchange_type   IN             SEEDOBJ_MFQUEUE.EXCHANGE_TYPE%TYPE,
           I_exchange_rate   IN             SEEDOBJ_MFQUEUE.EXCHANGE_RATE%TYPE)
RETURN BOOLEAN;
```

This function is called by either the COUNTRY or CURRENCY_RATES row trigger, and takes the message type and the table values (country_id for COUNTRY table and currency_code for CURRENCY_RATES table). It inserts a row into the message family queue along with the passed in values and the next sequence number from the message family sequence. The pub status will always be 'U' except for create messages, then it will be 'N'. It returns error codes and strings according to the standards of the application in which it is being implemented.

Function Level Description – GETNXT

```
PROCEDURE GETNXT
          (O_status_code     IN OUT         VARCHAR2,
           O_error_msg       IN OUT         RTK_ERRORS.RTK_TEXT%TYPE,
           O_message_type    IN OUT         VARCHAR2,
           O_message         IN OUT         RIB_OBJECT,
           O_bus_obj_id      IN OUT         RIB_BUSOBJID_TBL,
           O_routing_info    IN OUT         RIB_ROUTINGINFO_TBL,
           I_num_threads     IN             NUMBER DEFAULT 1,
           I_thread_val      IN             NUMBER DEFAULT 1)
```

The RIB calls GETNXT to get messages. It performs a cursor loop on the unpublished records on the SEEDOBJ_MFQUEUE table (PUB_STATUS = 'U'). It should only need to execute one loop iteration in most cases. For each record retrieved, GETNXT checks for records on the queue with a status of 'H'ospital. If there are any such records for the current business object, GETNXT raises an exception to send the current message to the Hospital.

The information from the SEEDOBJ_MFQUEUE and table is passed to PROCESS_QUEUE_RECORD. PROCESS_QUEUE_RECORD will build the Oracle Object message to pass back to the RIB. If PROCESS_QUEUE_RECORD does not run successfully, GETNXT raises an exception.

If any exception is raised in GETNXT, including the exception raised by an unsuccessful call to PROCESS_QUEUE_RECORD, HANDLE_ERRORS is called.

Function Level Description – PUB_RETRY

```
Procedure: PUB_RETRY
          (O_status_code     OUT            VARCHAR2,
           O_error_msg       OUT            VARCHAR2,
           O_message_type    IN  OUT        VARCHAR2,
           O_message         OUT            RIB_OBJECT,
           O_bus_obj_id      IN  OUT        RIB_BUSOBJID_TBL,
           O_routing_info    IN  OUT        RIB_ROUTINGINFO_TBL,
           I_REF_OBJECT      IN             RIB_OBJECT);
```

Same as GETNXT except:

It only loops for a specific row in the SEEDOBJ_MFQUEUE table. The record on SEEDOBJ_MFQUEUE must match the passed in sequence number (contained in the ROUTING_INFO).

Trigger Impact

Trigger Name: EC_TABLE_CNT_AIUDR.TRG

Trigger File Name: ec_table_cnt_aiudr.trg

Table: COUNTRY

This trigger will capture inserts/updates/deletes to the COUNTRY table and write data into the SEEDOBJ_MFQUEUE message queue.

Inserts

- Set the message type RMSMFM_SEEDOBJ.COUNTRY_ADD. Call ADDTOQ with all the country input parameters.

Updates

- Set the message type RMSMFM_SEEDOBJ.COUNTRY_UPD. Make sure that the country description has changed. If the description has changed during update, then call ADDTOQ with all the country input parameters.

Deletes

- Set the message type RMSMFM_SEEDOBJ. COUNTRY_DEL. Set the action type to ‘D’. Call ADDTOQ with all the country_id input parameter.

Trigger Name: EC_TABLE_CRT_AIUR.TRG

Trigger File Name: ec_table_crt_aiur.trg

Table: CURRENCY_RATES

This trigger will capture inserts/updates to the CURRENCY_RATES table and write data into the SEEDOBJ_MFQUEUE message queue.

Inserts

- Set the message type RMSMFM_SEEDOBJ.COUNTRY_ADD. Call ADDTOQ with all the currency_rates input parameters.

Updates

- Set the message type RMSMFM_SEEDOBJ.COUNTRY_UPD. Make sure that some column in the CURRENCY_RATES table has changed. If a column in the row has changed during update, then call ADDTOQ with all the currency_rates input parameters.

DTD

Here are the filenames that correspond with each message type. Please consult the RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
countrycre	Code Head Create Message	CountryDesc.dtd
countrymod	Code Head Modify Message	CountryDesc.dtd
countrydel	Code Head Delete Message	CountryRef.dtd
curratecre	Code Detail Create Message	CurrRateDesc.dtd
curratemod	Code Detail Modify Message	CurrRateDesc.dtd

Table Impact

TABLE	SELECT	INSERT	UPDATE	DELETE
SEEDOBJ _MFQUEUE	Yes	Yes	No	Yes

Store

Business Overview

RMS publishes data about stores in messages to the Oracle Retail Integration Bus (RIB). Other applications that need to keep their locations synchronized with RMS subscribe to these messages.

RMS publishes messages to the RIB for create, modify, and delete store events. These messages are triggered by insert/update/delete on the RMS STORE table and/or the ADDR table with module 'ST' (for store). The message is in a hierarchical structure, with store information at the header level and address information at the detail level. Because this is a low volume process, multi-threading capability is not supported. In addition, the system only publishes the current state of the store, not every change.

If multiple addresses are associated with a store, this message is designed with the assumption that RWMS and other integration subsystems only subscribe to the primary address of the primary address type.

For an additional explanation of virtual locations and the multi-channel operation of RMS, see the chapter "Organization Hierarchy Batch" in volume 1 of this RMS Operations Guide. For an explanation of finishers, see the section 'Partner' in the chapter "Publication Design" in this volume of the RMS Operations Guide.

Functionality Checklist

Description	RMS	RIB
RMS must publish store information		
Create new Publisher	X	X

Form Impact

None.

Business Object Records

Create the following business objects to assist the publishing process:

- Create a STORE_KEY_REC (in rmsmfm_stores.pls) that contains the functional keys to store_mfqueue publishing:

```
TYPE store_key_rec IS RECORD
               (STORE          STORE.STORE_TYPE%TYPE,
                ADDR_KEY       ADDR.ADDR_KEY%TYPE,
                STORE_TYPE     VARCHAR2 (1));
```

Package Impact

For Header Insert,Update, and Delete:

HDR_ADD

- Create a store with 1 or more addresses. HDR_ADD message must be added to the queue.
- Create a store with number of addresses > RIB_SETTINGS.MAX_DETAILS_TO_PUBLISH for store family. Test partial publishing of details in GETNXT.

HDR_UPD

- Update an existing store header record that has already been published. HDR_UPD message must be added to the queue.
- Update an existing store header record that has NOT been published yet. HDR_UPD message must be added to the queue.

HDR_DEL

- Delete an existing store header record that has already been published. HDR_DEL message must be added to the queue.
- Delete an existing store header record that has NOT been published yet. HDR_DEL message should NOT be added to the queue. In addition, all other message related to the store must be deleted from the queue. The store must also be deleted from STORE_PUB_INFO.

For Detail Insert, Update, and Delete:

DTL_ADD

- Add a new address to an existing store. DTL_ADD message must be added to the queue.

DTL _UPD

- Update an existing address of a store that has already been published. Any existing DTL_UPD message for the store/address in the queue must be deleted. The current DTL_UPD message for the store/address must be added to the queue.
- Update an existing address of a store that has NOT been published yet. Any existing DTL_UPD message for the store/address in the queue must be deleted. The current DTL_UPD message for the store/address does NOT need to be added to the queue.

DTL _DEL

- Delete an existing address of a store that has already been published. The address has also been published. Any existing message for the store/address in the queue must be deleted. The current DTL_DEL for the store/address must be added to the queue.
- Delete an existing address of a store that has already been published. The address has NOT been published yet. Any existing message for the store/address in the queue must be deleted. The current DTL_DEL for the store/address does NOT need to be added to the queue.

Package name: RMSMFM_STORE

Spec File Name: rmsmfm_stores.pls

Body File Name: rmsmfm_storeb.pls

Package Specification – Global Variables

```
FAMILY      CONSTANT      RIB_SETTINGS.FAMILY%TYPE   'STORE';
HDR_ADD     CONSTANT      VARCHAR2 (15)                      'storecre';
HDR_UPD     CONSTANT      VARCHAR2 (15)                      'storemod';
HDR_DEL     CONSTANT      VARCHAR2 (15)                      'storedel';
DTL_ADD     CONSTANT      VARCHAR2 (15)                      'storedtlcre';
DTL_UPD     CONSTANT      VARCHAR2 (15)                      'storedtlmode';
DTL_DEL     CONSTANT      VARCHAR2 (15)                      'storedtldel';
```

Function Level Description – ADDTOQ

```
Function: ADDTOQ
              (O_error_msg                OUT           VARCHAR2,
               I_message_type             IN            VARCHAR2,
               I_store_key_rec            IN            STORE_KEY_REC,
               I_addr_publish_ind         IN            ADDR.PUBLISH_IND%TYPE)
```

This public function puts a store message on STORE_MFQUEUE for publishing to the RIB. It is called from both store trigger and address trigger. The I_functional_keys will contain store and, optionally, addr_key.

There are some tasks relating to streamlining the queue clean up process that need to occur in ADDTOQ. The goal is to have at most one record on the queue for a given store up until its initial publication.

- For header level insert messages (HDR_ADD), insert a record in the STORE_PUB_INFO table. The published flag should be set to 'N'. The STORE_TYPE column should have a value of 'P'hysical or 'V'irtual. Because this is a low volume business transaction, the store publisher will not provide multi-threading capability. Therefore, thread value does NOT need to be calculated.
- For HDR_UPD, DTL_ADD, DTL_UPD, DTL_DEL messages, do NOT need to be added to the queue until the business object (that is, store) has been initially published (STORE_PUB_INFO.published = 'N'). This is because when the store does get published for the first time (HDR_ADD), the current state of the header and details is queried and published to the RIB.
- For header level delete messages (HDR_DEL), if the business object (that is, store) has NOT been initially published (STORE_PUB_INFO.published = 'N'), delete the record in STORE_PUB_INFO. Otherwise, delete every record in the queue for the business object, and add the delete record to the queue.
- If the business object has been initially published (STORE_PUB_INFO.published = 'Y'), for detail level messages deletes (DTL_DEL), the system only needs one (the most recent) record per detail in the STORE_MFQUEUE. Delete any previous records that exist on the STORE_MFQUEUE for the record that has been passed.

- If the business object has been initially published (STORE_PUB_INFO.published = 'Y'), for detail level messages updates (DTL_UPD), the system only needs one DTL_UPD (the most recent) record per detail in the STORE_MFQUEUE. Delete any previous DTL_UPD records that exist on the STORE_MFQUEUE for the record that has been passed. The system does not want to delete any detail inserts that exist on the queue for the detail. The system needs to ensure subscribers are not passed a detail modification message for a detail that they do not yet have.
- For all message types except header level inserts (HDR_ADD), insert a record into the STORE_MFQUEUE. One exception is that if the publish_ind on the detail record table (addr) is 'N', do not add the DTL_DEL message to the queue. The logic here is that if the detail line has never been published before, subscribers will not need to delete the detail line that they do not yet have.

Function Level Description – GETNXT

```
Procedure: GETNXT
               (O_status_code        OUT    VARCHAR2,
                O_error_msg          OUT    VARCHAR2,
                O_message_type       OUT    VARCHAR2,
                O_message            OUT    RIB_OBJECT,
                O_bus_obj_id         OUT    RIB_BUSOBJID_TBL,
                O_routing_info       OUT    RIB_ROUTINGINFO_TBL,
                I_num_threads        IN     NUMBER DEFAULT 1,
                I_thread_val         IN     NUMBER DEFAULT 1)
```

This public procedure is called from the RIB to get the next messages. It performs a cursor loop on the unpublished records on the STORE_MFQUEUE table (PUB_STATUS = 'U'). It should only need to execute one loop iteration in most cases. For each record retrieved, GETNXT gets the following:

1. A lock of the queue table for the current business object (that is, store). The lock is obtained by calling the function LOCK_THE_BLOCK. If there are any records on the queue for the current business object that are already locked, the current message is skipped and picked up again in the next loop iteration.
2. A check for records on the queue with a status of 'H'ospital. If there are any such records for the current business object, GETNXT raises an exception to send the current message to the Hospital.
3. Get the published indicator from the STORE_PUB_INFO table.
4. Call PROCESS_QUEUE_RECORD with the current business object.

The loop will need to execute more than one iteration for the following cases:

1. When a header delete message exists on the queue for a business object that has not been initially published. In this case, remove the header delete message from the queue and loop again. ADDTOQ will delete all messages for the business object upon header delete if the business object has not been initially published.
2. The queue is locked for the current business object. This can occur because ADDTOQ, which is called from the triggers, deletes from the queue table for DTL_UPD, DTL_DEL, and HDR_DEL messages.

The information from the STORE_MFQUEUE and STORE_PUB_INFO table is passed to PROCESS_QUEUE_RECORD. PROCESS_QUEUE_RECORD will build the Oracle Object message to pass back to the RIB. If PROCESS_QUEUE_RECORD does not run successfully, GETNXT raises an exception.

If any exception is raised in GETNXT, including the exception raised by an unsuccessful call to PROCESS_QUEUE_RECORD, HANDLE_ERRORS is called.

Function Level Description – PUB_RETRY

```
Procedure: PUB_RETRY
              (O_status_code        OUT          VARCHAR2,
               O_error_msg          OUT          VARCHAR2,
               O_message            OUT          RIB_OBJECT,
               O_message_type       IN OUT       VARCHAR2,
               O_bus_obj_id         IN OUT       RIB_BUSOBJID_TBL,
               O_routing_info       IN OUT       RIB_ROUTINGINFO_TBL)
```

This public procedure performs the same tasks as GETNXT except that it only loops for a specific row in the STORE_MFQUEUE table. The record on STORE_MFQUEUE must match the passed in sequence number (contained in the ROUTING_INFO).

Function Level Description – PROCESS_QUEUE_RECORD (local)

This private function controls the building of Oracle Objects (DESC or REF) given the business transaction's key values and a message type. It contains all of the shared processing between GETNXT and PUB_RETRY. (Note: message_type of HDR_ADD can potentially be changed to a DTL_ADD in PROCESS_QUEUE_RECORD.)

If the message type is a header delete (HDR_DEL)

- I_hdr_published should never be 'N' (not published), because in that case the HDR_DEL message would never be added to the queue based on ADDTOQ processing.
- Call BUILD_HEADER_OBJECT to build the REF Oracle Object to publish to the RIB.
- Delete both STORE_MFQUEUE and STORE_PUB_INFO for the business object.

Otherwise, if check I_hdr_published is either 'N' (not published) or 'I' (in progress)

- If I_hdr_published is 'N', the message type is a header create (HDR_ADD). If I_hdr_published is 'I', change the message type from a header create (HDR_ADD) to a detail add (DTL_ADD), because this is the situation where some of the details are published but not all due to MAX_DETAILS_TO_PUBLISH. Thus, a header create message for the current business object should have already been published.
- Call MAKE_CREATE to build the DESC Oracle Object to publish to the RIB. This will also take care of any STORE_MFQUEUE deletes, updating STORE_PUB_INFO.PUBLISHED to 'Y' or 'I', and bulk updating addr.publish_ind to 'Y' for those detail rows that have been published.

Otherwise,

If the record from STORE_QUEUE table is a header update (HDR_UPD)

- Call BUILD_HEADER_OBJECT to build the DESC Oracle Object to publish to the RIB.
- Update STORE_PUB_INFO to published = 'Y'.
- Delete the record from the STORE_MFQUEUE table.

If the record from STORE_QUEUE table is a detail add or update (DTL_ADD, DTL_UPD)

- Call BUILD_DETAIL_CHANGE_OBJECTS to build the DESC Oracle Object to publish to the RIB. This will also take care of any STORE_MFQUEUE deletes and updates of publish_ind on ADDR.

If the record from STORE_QUEUE table is a detail delete (DTL_DEL)

- Call BUILD_DETAIL_DELETE_OBJECTS to build the REF Oracle Object to publish to the RIB. This will also take care of any STORE_MFQUEUE deletes.

Function Level Description – MAKE_CREATE (local)

This private function is used to create the Oracle Object for the initial publication of a business transaction. I_business_object contains the store header key values (store). I_rowid is the rowid of the store_mfqueue row fetched from GETNXT.

- Call BUILD_HEADER_OBJECT to get a header level Oracle Object.
- Call BUILD_DETAIL_OBJECTS to get a table of detail level Oracle objects and a table of STORE_MFQUEUE rowids to delete with and a table of ADDR rowids to update publish_ind with.
- Update STORE_PUB_INFO.published to 'Y' or 'I' depending on if all details are published.
- Delete records from the STORE_MFQUEUE for all rowids returned by BUILD_DETAIL_OBJECTS. Deletes are done by rowids instead of business transaction keys to ensure that nothing is deleted off the queue that has not been published.
- If the entire business transaction was added to the Oracle Object, also delete the STORE_MFQUEUE record that was picked up by GETNXT. If the entire business transaction was NOT published, the system needs to leave something on the STORE_MFQUEUE to ensure that the rest of it is picked up by the next call to GETNXT.
- Update ADDR.publish_ind to 'Y' by ADDR rowids returned from BUILD_DETAIL_OBJECTS.
- The header and detail level Oracle Objects are combined and returned.

Function Level Description – BUILD_HEADER_OBJECT (local)

This private function accepts store header key value (store), builds and returns a header level DESC Oracle Object.

This overloaded private function accepts store header key value (store), builds and returns a header level REF Oracle Object.

Function Level Description – BUILD_DETAIL_OBJECTS (local)

The private function is responsible for building detail level DESC Oracle Objects. It builds as many detail Oracle Object as it can given the passed in message type and business object keys (store).

Call API_LIBRARY.GET_RIB_SETTINGS to get the MAX_DETAILS_TO_PUBLISH for the store family.

If the function is being called from MAKE_CREATE (I_message_type would be NULL):

Select any unpublished ADDR detail records for the business transaction (based on publish_ind on ADDR). Create Oracle Objects for details that are selected by calling BUILD_SINGLE_DETAIL.

- Ensure that no more than MAX_DETAILS_TO_PUBLISH records are put into Oracle Objects.
- Return a table of store_mfqueue rowids for each message that is placed into the Oracle Object.
- Return a table of addr rowids for each detail that is placed into the Oracle Object.

If the function is **not** being called from MAKE_CREATE (I_message type will NOT be NULL):

Select any details on the STORE_MFQUEUE that are for the same business object and for the same message type. Create Oracle Objects for details that are selected by calling BUILD_SINGLE_DETAIL.

- Ensure that no more than MAX_DETAILS_TO_PUBLISH records are put into Oracle Objects.
- If the message type is a detail create (DTL_ADD), as the detail info is placed into the Oracle Object, ensure that the corresponding addr rowid is added to addr rowids table for return. These rowids are used to update ADDR.publish_ind to 'Y'.
- Return a table of store_mfqueue rowids for each message that is placed into the Oracle Object.

A concern here is making sure that the system does not delete information from the queue table that has not been published. For this reason, the system does our deletes by ROWID. The system also tries to get everything in the same cursor. This should ensure that the message published matches the deletes performed from the STORE_MFQUEUE table regardless of trigger execution during GETNXT calls.

Function Level Description – BUILD_SINGLE_DETAIL (local)

This private function takes in an address record and builds a detail level Oracle Object. Also find out if the address is the primary address of the primary address type and set the DESC Oracle Object accordingly.

Function Level Description – BUILD_DETAIL_CHANGE_OBJECTS (local)

This private function builds a DESC Oracle Object to publish to the RIB for detail create and detail update messages (DTL_ADD, DTL_UPD). I_business_obj contains the header level key values (store).

- Call BUILD_HEADER_OBJECT to build the header level DESC Oracle Object.
- Call BUILD_DETAIL_OBJECTS to build the detail level DESC Oracle Objects.
- Bulk update addr.publish_ind to 'Y' for the addr rowids returned from BUILD_DETAIL_OBJECTS.
- Bulk delete from store_mfqueue for the store_mfqueue rowids returned from BUILD_DETAIL_OBJECTS.

Function Level Description – BUILD_DETAIL_DELETE_OBJECTS (local)

This private function builds a REF Oracle Object to publish to the RIB for detail delete messages (DTL_DEL). I_business_obj contains the header level key values (store).

- Call API_LIBRARY.GET_RIB_SETTINGS to get the MAX_DETAILS_TO_PUBLISH for the store family.
- Call BUILD_HEADER_OBJECT to build the REF Oracle Object to publish to the RIB.
- Perform a cursor for loop on STORE_MFQUEUE and build as many detail REF Oracle Objects as possible without exceeding the MAX_DETAILS_TO_PUBLISH.
- For each detail, also find out if the address is a primary address of the primary address type, and set the REF Oracle Objects accordingly.
- Bulk delete from store_mfqueue for the store_mfqueue rowids queried.

Function Level Description – LOCK_THE_BLOCK (local)

This private function locks all queue records for the current business object (store). This is to ensure that GETNXT and PUB_RETRY do not wait on any business processes that currently have the queue table locked and have not committed. This can occur because ADDTOQ, which is called from the triggers, deletes from the queue table for DTL_UPD, DTL_DEL, and HDR_DEL messages.

Function Level Description – HANDLE_ERRORS (local)

This private procedure is called from GETNXT and PUB_RETRY when an exception is raised. I_seq_no is the sequence number of the driving STORE_MFQUEUE record. I_function_keys contains detail level key values (store, addr_key).

If the error is a non-fatal error, HANDLE_ERRORS passes the sequence number of the driving STORE_MFQUEUE record back to the RIB in the ROUTING_INFO. It sends back a status of 'H'ospital to the RIB as well. It then updates the status of the queue record to 'H'ospital, so that it will not get picked up again by the driving cursor in GETNXT.

If the error is a fatal error, a status of 'E'rror is returned to the RIB. The error is considered non-fatal if no DML has occurred yet. Whenever DML has occurred, then the global variable LP_error_status is flipped from 'H'ospital to 'E'rror.

Trigger Impact

Trigger Name: EC_TABLE_STR_AIUDR.TRG (new)

Trigger File Name: ec_table_str_aiudr.trg (new)

Table: STORE

This is the trigger on the STORE table that will capture Inserts, Updates, and Deletes.

Inserts

- Send the I_store_key_rec, I_addr_publish_ind to the ADDTOQ procedure in the MFM with the message type RMSMFM_STORE.HDR_ADD.

Updates

- Send the I_store_key_rec, I_addr_publish_ind to the ADDTOQ procedure in the MFM with the message type RMSMFM_STORE.HDR_UPD.

Deletes

- Send the I_store_key_rec, I_addr_publish_ind to the ADDTOQ procedure in the MFM with the message type RMSMFM_STORE.HDR_DEL.

In all these cases, build the function keys for ADDTOQ with store.

Trigger Name: EC_TABLE_ADR_AIUDR.TRG (mod)

Trigger File Name: ec_table_adr_aiudr.trg (mod)

Table: ADDR

This is the trigger on the ADDR table that will capture Inserts, Updates, and Deletes of module type ‘ST’. (Note: It also handles module types supplier, partner and warehouse.)

Inserts

- Send the I_store_key_rec.store, I_store_key_rec.addr_key, I_addr_publish_ind to the ADDTOQ procedure in the MFM with the message type RMSMFM_STORE.DTL_ADD.

Updates

- Send the I_store_key_rec.store, I_store_key_rec.addr_key, I_addr_publish_ind to the ADDTOQ procedure in the MFM with the message type RMSMFM_STORE.DTL_UPD.

Deletes

- Send the I_store_key_rec.store, I_store_key_rec.addr_key, I_addr_publish_ind to the ADDTOQ procedure in the MFM with the message type RMSMFM_STORE.DTL_DEL.

In all these cases, build the function keys for ADDTOQ with KEY_VALUE_1 and KEY_VALUE_2, which represent store and addr_key respectively.

DTD

Here are the filenames that correspond with each message type. Please consult the RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
StoreCre	Store Create Message	StoreDesc.dtd
StoreMod	Store Modify Message	StoreDesc.dtd
StoreDel	Store Delete Message	StoreRef.dtd

Table Impact

TABLE	SELECT	INSERT	UPDATE	DELETE
STORE_PUB_INFO	Yes	Yes	Yes	Yes
ADDR	Yes	No	Yes	No
STORE_MFQUEUE	Yes	Yes	Yes	Yes
ADD_TYPE_MODULE	Yes	No	No	No
STORE	Yes	No	No	No

Design Assumptions

It is not possible for a detail trigger to accurately know the status of a header table. In order for the detail triggers to accurately know when to add a message to the queue, RMS should **not** allow approval of a business object while detail modifications are being made.

It is not possible for a header trigger or a detail trigger to know the status of anything modified by GETNXT. If a header trigger or detail trigger is trying to delete queue records that GETNXT currently has locked, it will have to wait until GETNXT is finished and removes the lock. It is assumed that this time will be fairly short (at most 2-3 seconds.) It is also assumed that this will occur rarely, as it involves updating/deleting detail records on a business object that has already been approved. This also has to occur at the same time GETNXT is processing the current business object.

Push off all DML statements as late as possible. Once DML statements have taken place, any error becomes a fatal error rather than a hospital error.

Transfers

Business Overview

A transfer is a movement of stock on hand from one stockholding location within the company to another. For more general information on transfers, see the chapter "Transfers and RTV Batch" in volume 1 of this RMS Operations Guide.

The transfer publication processing publishes transfers in 'Approved' status.

Transfers consist of header level information in which source and destination locations are specified, and detail information regarding what items and how much of each item is to be transferred. Both of the transfer tables, TSFHEAD and TSFDETAIL, include triggers that track inserts, deletes, and modifications. These triggers insert or update into TSF_MFQUEUE or TRANSFERS_PUB_INFO tables. The transfer family manager is responsible for pulling transfer information from this queue and sending it to the external system(s) at the appropriate time and in the correct order.

The transfer messages that are published by the family manager vary. A complete message including header information, detail information, and component ticketing information (if applicable) is created when a transfer is approved. When the transfer is unapproved, the RIB processes it as a TransferDel message when publishing it to external systems. When the transfer is re-approved, the transfer is processed as a new transfer for publishing.

Context information is included at the header level. The context_type defines the business reason for the transfer, thus allowing users to distinguish one form of transfer from another. The context_value further identifies a specific context_type. For example, when the context of a transfer is promotion (that is, when the transfer is being created to support an RPM promotion), the ID of the promotion being supported is attached to the transfer as well, at the header level.

Functionality Checklist

Description		RMS	RIB
RMS must publish Transfers information			
	Create new Publisher	X	X

Form Impact

None.

Business Object Records

Create the following table types in the RMSMFM_TRANSFERs package:

TYPE rowid_TBL is table of ROWID INDEX BY BINARY_INTEGER;

Package Impact

Create Header

1. Prerequisites: None.
2. Activity Detail: The first step to creating a transfer is creating the header level information.
3. Messages: When a transfer is created, a record is inserted into TRANSERS_PUB_INFO table and is not published onto the queue until the transfer has been approved.

Approve

1. Prerequisites: A transfer must exist and have at least one detail before it can be approved.
2. Activity Detail: Approving a transfer changes the status of the transfer. This change in status signifies the first time systems external to RMS will have an interest in the existence of the transfer, so this is the first part of the life cycle of a transfer that is published.
3. Messages: When a transfer is approved, a "TransferHdrMod" message is inserted into the queue with the appr_ind on the queue set to 'Y' signifying that the transfer was approved. The family manager uses this indicator to create a hierarchical message containing a full snapshot of the transfer at the time the message is published.

Modify Header

1. Prerequisites: The transfer header can only be modified when the status is **not** approved. Once the transfer is approved, the only fields that are modifiable are the status field and the comments field.
2. Activity Detail: The user is allowed to modify the header but only certain fields at certain times. If a transfer is in input status the 'to and from' locations may be modified until details have been added. Once details have been added, the locations are disabled. The freight code is modifiable until the transfer has been approved. Comments can be modified at any time.
3. Messages: When the status of the header is either changed to 'C'losed or 'A'pproved, a message (TransferHdrMod) is inserted into the queue. (Look above at Approve activity and below at Close activity for further details).

Create Details

1. Prerequisites: A transfer header record must exist before transfer details can be created.
2. Activity Detail: The user is allowed to add items to a transfer, but only until it has been approved. Once a transfer has been approved, details can longer be added.
3. Messages: No messages are created on the queue until the transfer is approved.

Modify Details

1. Prerequisites: Only modifications to transfer quantities are sent to the queue, and only when the transfer quantity is decreased manually, and not because of an increase in cancelled quantity will it be sent to the queue.
2. Activity Detail: The user is allowed to change transfer quantities provided they are not reduced below those already shipped. The transfer quantity can also be decreased by an increase in the cancelled quantity, which is always initiated by the external system. This change, then, would be of no interest to the external system because it was driven by it.
3. Messages: No messages are created on the queue until the transfer is approved.

Delete Details

1. Prerequisites: Only a detail that has not been shipped may be deleted, and it cannot be deleted if it is currently being worked on by an external system. A user is not allowed to delete details from a closed transfer.
2. Activity Detail: A user is allowed to delete details from a transfer but only if the item has not been shipped.
3. Messages: No messages are created on the queue until the transfer is approved.

Close

1. Prerequisites: A transfer must be in shipped status before it can be closed, and it cannot be in the process of being worked on by an external system.
2. Activity Detail: Closing a transfer changes the status, which prevents any further modifications to the transfer. When a transfer is closed, a message is published to update the external system(s) that the transfer has been closed and no further work (in RMS) is performed on it.
3. Messages: Closing a transfer queues a “TransferHdrMod” request. This is a flat message containing a snapshot of the transfer header information at the time the message is published.

Delete

1. Prerequisites: A transfer can only be deleted when it is still in approved status or when it has been closed.
2. Activity Detail: Deleting a transfer removes it from the system. External systems are notified by a published Delete message that contains the number of the transfer to be deleted.
3. Message: When a transfer is deleted, a “TransferDel”, which is a flat notification message, is queued.

Package Name: RMSMFM_TRANSFERS

Spec File Name: rmsmfm_transferss.pls

Body File Name: rmsmfm_transfersb.pls

Package Specification – Global Variables

```
FAMILY      VARCHAR2(64) := 'transfers';

HDR_ADD     VARCHAR2(64) := 'TransferCre';
HDR_UPD     VARCHAR2(64) := 'TransferHdrMod';
HDR_DEL     VARCHAR2(64) := 'TransferDel';
HDR_UNAPRV  VARCHAR2(64) := 'TransferUnapp';
DTL_ADD     VARCHAR2(64) := 'TransferDtlCre';
DTL_UPD     VARCHAR2(64) := 'TransferDtlMod';
DTL_DEL     VARCHAR2(64) := 'TransferDtlDel';
```

Function Level Description – ADDTOQ

```
FUNCTION ADDTOQ
               (O_error_mesage       OUT    VARCHAR2,
                I_message_type       IN     VARCHAR2,
                I_tsf_no             IN     tsfhead.tsf_no%TYPE,
                I_tsf_type           IN     tsfhead.tsf_type%TYPE,
                I_tsf_head_status    IN     tsfdetail.status%TYPE,
                I_item               IN     tsfdetail.item%TYPE,
                I_publish_ind        IN     tsfdetail.publish_ind%TYPE)
```

This function is called by both the tsfhead trigger and the tsfdetail trigger, the EC_TABLE_THD_AIUDR and EC_TABLE_TDT_AIUDR respectively.

- Book transfers, non-sellable transfers and externally generated transfers (except for delete messages) are never published to external systems.
- For header level insert messages (HDR_ADD), insert a record in the TRANSFERS_PUB_INFO table. The published flag should be set to ‘N’. The correct thread for the Business transaction should be calculated and written. Call API_LIBRARY.RIB_SETTINGS to get the number of threads used for the publisher. Using the number of threads, and the Business object id, calculate the thread value.
- For all records except header level inserts (HDR_ADD), the thread_no and initial_approval_ind should be queried from the TRANSFERS_PUB_INFO table.
- If the business transaction has not been approved (initial_approval_ind = ‘N’) and the triggering message is one of DTL_ADD, DTL_UPD, DTL_DEL, HDR_DEL, no processing should take place and the function should exit.
- For detail level message deletes (DTL_DEL), only need one (the most recent) record per detail in the TSF_MFQUEUE is required. Delete any previous records that exist on the TSF_MFQUEUE for the record that has been passed. If the publish_ind is ‘N’, do not add the DTL_DEL message to the queue.
- For detail level message updates (DTL_UPD), only need one DTL_UPD (the most recent) record per detail in the TSF_MFQUEUE is required. Delete any previous DTL_UPD records that exist on the TSF_MFQUEUE for the record that has been passed. The system does not want to delete any detail inserts that exist on the queue for the detail. It needs to ensure subscribers have not passed a detail modification message for a detail that they do not yet have.

- For header level delete messages (HDR_DEL), delete every record in the queue for the Business transaction.
- For header level update message (HDR_UPD), update the TRANSFERS_PUB_INFO.INITIAL_APPROVAL_IND to 'Y' if the Business transaction is in approved status.
- For all records except header level inserts (HDR_ADD), insert a record into the TSF_MFQUEUE.

It returns a status code of API_CODES.SUCCESS if successful, API_CODES.UNHANDLED_ERROR if not.

Function Level Description – GETNXT

```
PROCEDURE GETNXT
               (O_status_code        OUT    VARCHAR2,
                O_error_msg          OUT    VARCHAR2,
                O_message_type       OUT    VARCHAR2,
                O_message            OUT    RIB_OBJECT,
                O_bus_obj_id         OUT    RIB_BUSOBJID_TBL,
                O_routing_info       OUT    RIB_ROUTINGINFO_TBL,
                I_num_threads        IN     NUMBER DEFAULT 1,
                I_thread_val         IN     NUMBER DEFAULT 1)
```

The RIB calls GETNXT to get messages. It performs a cursor loop on the unpublished records on the TSF_MFQUEUE table (PUB_STATUS = 'U'). It should only need to execute one loop iteration in most cases. For each record retrieved, GETNXT gets the following:

1. A lock of the queue table for the current Business object. The lock is obtained by calling the function LOCK_THE_BLOCK. If there are any records on the queue for the current Business object that are already locked, the current message is skipped.
2. The published indicator from the TRANSFERS_PUB_INFO table.
3. A check for records on the queue with a status of 'H'ospital. If there are any such records for the current Business object, GETNXT raises an exception to send the current message to the Hospital.

The loop executes more than one iteration for the following cases

1. When a header delete message exists on the queue for a business object that has not been initially published. In this case, simply remove the header delete message from the queue and loop again.
2. A detail delete message exists on the queue for a detail record that has not been initially published. In this case, simply remove the detail delete message from the queue and loop again.
3. The queue is locked for the current Business object

The information from the TSF_MFQUEUE and TRANSFERS_PUB_INFO table is passed to PROCESS_QUEUE_RECORD. PROCESS_QUEUE_RECORD builds the Oracle Object message to pass back to the RIB. If PROCESS_QUEUE_RECORD does not run successfully, GETNXT raises an exception.

If any exception is raised in GETNXT, including the exception raised by an unsuccessful call to PROCESS_QUEUE_RECORD, HANDLE_ERRORS is called.

Function Level Description – PUB_RETRY

Same as GETNXT except:

It only loops for a specific row in the TSF_MFQUEUE table. The record on TSF_MFQUEUE must match the passed in sequence number (contained in the ROUTING_INFO).

Function Level Description – PROCESS_QUEUE_RECORD (local)

This function controls the building of Oracle Objects given the business transaction's key values and a message type. It contains all of the shared processing between GETNXT and PUB_RETRY.

If the message type is HDR_DEL or HDR_UNAPRV and it has not been published:

- Call DELETE_QUEUE_REC to delete the record from TSF_MFQUEUE.

If the message type is HDR_DEL and the record has been published:

- Generate a "flat" file to be sent to the RIB. Delete from TRANSFER_PUB_INFO and call DELETE_QUEUE_REC to delete from the queue.

If the message type is HDR_UNAPRV"

- Process it just like a hdr_del except the published indicator on TRANSFERS_PUB_INFO is set to 'N'.

If the message type is HDR_ADD or DTL_ADD:

- Call MAKE_CREATE to publish the entire transfer.

If the record from TSF_MFQUEUE table is HDR_UPD:

- Call BUILD_HEADER_OBJECT to build the Oracle Object to publish to the RIB and delete from the queue.

If the record from TSF_MFQUEUE table is DTL_ADD or DTL_UPD:

- Call BUILD_HEADER_OBJECT and BUILD DETAIL_CHANGE_OBJECTS to build the Oracle Object to publish to the RIB.

If the record from TSF_MFQUEUE table is a detail delete (DTL_DEL):

- Call BUILD HEADER_OBJECT and BUILD_DETAIL_DELETE_OBJECTS to build the Oracle Object to publish to the RIB.

Function Level Description – MAKE_CREATE (local)

This function is used to create the Oracle Object for the initial publication of a business transaction. It combines the current message and all previous messages with the same key in the queue table to create the complete hierarchical message. It first creates a new message with the hierarchical document type. It then gets the header create message and adds it to the new message. The remainder of this procedure gets each of the details grouped by their document type and adds them to the new message. When it is finished creating the new message, it deletes all the records from the queue with a sequence number less than or equal to the current records sequence number. This new message is passed back to the bus. The MAKE_CREATE function will not be called unless the appr_ind on the queue is 'Y'es (meaning the transfer has been approved, and it is ready to be published for the first time to the external system(s)).

Function Level Description – BUILD_HEADER_OBJECT (local)

Accepts header key values, performs necessary lookups, builds and returns a header level Oracle Object.

Function Level Description – BUILD_DETAIL_OBJECTS (local)

This function is responsible for fetching the detail info and ticket type to be sent to RWMS. The logic that gets the detail info as well as the ticket type was separated to remove the primary key constraint.

Function Level Description – BUILD_SINGLE_DETAIL (local)

Accept inputs and build a detail level Oracle Object. Perform any lookups needed to complete the Oracle Object.

Function Level Description – GET_RETAIL(local)

Gets the price and selling unit of measure (UOM) of the item.

Function Level Description – GET_GLOBALS(local)

Get all the system options and variables needed for processing.

Function Level Description – GET_TSF_ENTITIES(local)

Get the to and from location entities for the transfer.

Function Level Description – BUILD_DETAIL_CHANGE_OBJECTS (local)

Call BUILD_DETAIL_OBJECT to publish the record. Update TSFDETAIL.publish_ind to 'Y' and delete the record from TSF_MFQUEUE.

Function Level Description – BUILD_DETAIL_DELETE_OBJECTS (local)

Either pass in a header level ref Oracle Object or build a header level ref Oracle Object.

Perform a cursor for loop on TSF_MFQUEUE and build as many detail ref Oracle Objects as possible without exceeding the MAX_DETAILS_TO_PUBLISH.

Delete from TSF_MFQUEUE when done.

Function Level Description – LOCK_THE_BLOCK (local)

This function locks all queue records for the current business object. This is to ensure that GETNXT does not wait on any business processes that currently have the queue table locked and have not committed. This can occur because ADDTOQ, which is called from the triggers, deletes from the queue table for DTL_UPD, DTL_DEL, and HDR_DEL messages.

Function Level Description – LOCK_THE_BLOCK (local)

Lock the transfer details before updating the publish_ind on TSFDETAIL.

Function Level Description – DELETE_QUEUE_REC (local)

This procedure deletes a specific record from TSF_MFQUEUE. It deletes based on the sequence number passed in.

Function Level Description – HANDLE_ERRORS (local)

HANDLE_ERRORS is called from GETNXT and PUB_RETRY when an exception is raised. The function was updated to conform with the changes made to the ADDTOQ function.

Trigger Impact

Create a trigger on the TSFHEAD and TSFDETAIL to capture Inserts, Updates, and Deletes.

Trigger Name: EC_TABLE_THD_AIUDR.TRG

Trigger File Name: ec_table_thd_aiudr.trg

Table: TSFHEAD

Inserts

- Send the tsf_no and tsf_type level info to the ADDTOQ procedure in the MFM with the message type RMSMFM_Transfers.HDR_ADD.

Updates

- Send the tsf_no and tsf_type level info to the ADDTOQ procedure in the MFM with the message type RMSMFM_Transfers.HDR_UPD.

Deletes

- Send the tsf_no and tsf_type level info to the ADDTOQ procedure in the MFM with the message type RMSMFM_Transfers.HDR_DEL.

Trigger Name: EC_TABLE_TDT_AIUDR.TRG

Trigger File Name: ec_table_tdt_aiudr.trg

Table: TSFDETAIL

Inserts

- Send the tsf_no and item level info to the ADDTOQ procedure in the MFM with the message type RMSMFM_Transfers.DTL_ADD

Updates

- Send the tsf_no and item level info to the ADDTOQ procedure in the MFM with the message type RMSMFM_Transfers.DTL_UPD.

Deletes

- Send the tsf_no and item level info to the ADDTOQ procedure in the MFM with the message type RMSMFM_Transfers.DTL_DEL.

DTD

Here are the filenames that correspond with each message type. See Oracle Retail Integration Bus documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
TransferCre	Transfer Create Message	TsfDesc.dtd
TransferHdrMod	Transfer Modify Message	TsfDesc.dtd
TransferDel	Transfer Delete Message	TsfRef.dtd
TransferDtlCre	Transfer Detail Create Message	TsfDtl.dtd
TransferDtlMod	Transfer Detail Modify Message	TsfDtl.dtd
TransferDtlDel	Transfer Detail Delete Message	TsfDtlRef.dtd

Table Impact

TABLE	SELECT	INSERT	UPDATE	DELETE
TRANSFERS_PUB_INFO	Yes	Yes	Yes	Yes
TSF_MFQUEUE	Yes	Yes	Yes	Yes
TSF_DETAIL	Yes	No	Yes	No
TSF_HEAD	Yes	No	No	No
WH	Yes	No	No	No
ORDCUST	Yes	No	No	No
CUSTOMER	Yes	No	No	No
ITEM_LOC	Yes	No	No	No
ITEM_MASTER	Yes	No	No	No
ITEM_TICKET	Yes	No	No	No
V_PACKSKU_QTY	Yes	No	No	No
CODE_DETAIL	Yes	No	No	No
SYSTEM_OPTIIONS	Yes	No	No	No
RIB_SETTINGS	Yes	No	No	No

Design Assumptions

- After a transfer has been approved, Oracle Retail assumes the freight code of the transfer (on the TSFHEAD table) cannot be updated.
- One of the primary assumptions in the current approach is that ease of code will outweigh performance considerations. It is hoped that the ‘trickle’ nature of the flow of data will decrease the need to dwell on performance issues and instead allow developers to code in the easiest and most straight forward manner.
- The adaptor is only set up to call stored procedures, not stored functions. Any public program then needs to be a procedure.

UDA

Business Overview

RMS publishes messages about user-defined attributes (UDAs) to the Oracle Retail Integration Bus (RIB). UDAs provide a method for defining attributes and associating the attributes with specific items, items on an item list, or items in a specific department, class, or subclass. UDAs are useful for information and reporting purposes. Unlike traits or indicators, UDAs are not interfaced with external systems. UDAs do not have any programming logic associated with them. UDA messages are specific to basic UDA identifiers and values defined in RMS. The UDAs can be displayed in one or more of three formats: Dates, Freeform Text, or a List of Values (LOV).

The created messages in the XML builder adds the messages to the UDA_MFQUEUE table which must be published in the same order as they occur in the RMS database.

For related UDA information, see the section 'Item' in the chapter "Publication Design" in this volume of this RMS Operations Guide.

Functionality Checklist

Description	RMS	RIB
RMS must publish UDA information		
Create new Publisher	X	X

Form Impact

None.

Business Object Records

None.

Package Impact

Create UDA

1. Prerequisites: UDA does not already exist.
2. Activity Detail: Any change to the UDA table inserts a UDAHdrCre message_type record on the UDA_MFQUEUE table.
3. Messages: The UDADesc message is created. It is a flat, synchronous message containing a full snapshot of the UDA at the time the message is published.

Modify UDA

1. Prerequisites: UDA exists.
2. Activity Detail: Any change to the UDA table inserts a UDAHdrMod message_type record on the UDA_MFQUEUE table.
3. Messages: The UDADesc message is created. It is a flat, synchronous message containing a full snapshot of the UDA at the time the message is published.

Create UDA_Values

1. Prerequisites: A UDA already exists but the uda_value does not exist.
2. Activity Detail: Any change to the UDA_VALUES table inserts a record to the UDA_VALUES table. A UDAValCre message type record is also inserted on the UDA_MFQUEUE table. A foreign key to the UDA table checks the existence of the UDA the value is created to supplement.
3. Messages: A UDAValDesc message type is created. It is a hierarchical, synchronous message containing a snapshot of the UDA_VALUES table at the time the message is published.

Modify UDA_Values

1. Prerequisites: UDA and UDA_value exist.
2. Activity Detail: Any change to the UDA_VALUES table updates a record to the UDA_VALUES table. A UDAValMod message type record is also inserted on the UDA_MFQUEUE table. A foreign key from the UDA_VALUES table to the UDA table checks the existence of the UDA the value is supplements.
3. Messages UDAValDesc message is created. It is a flat, synchronous message containing a snapshot of the UDA_VALUES table at the time the message is published.

Delete UDA_Values

1. Prerequisites: UDA_value exists.
2. Activity Detail: Deleting a UDA_value removes it from the UDA_VALUES table and inserts a UDAValDel row to the UDA_MFQUEUE table.
3. Message: A UDAValRef message is created. It is a flat, synchronous message containing the primary key with which the external systems can remove it from their systems.

Delete UDA

1. Prerequisites: UDA exists and a UDA_VALUE may or may not exist.
2. Activity Detail: Deleting a UDA removes it from the UDA table and inserts a UDAHdrDel row to the UDA_MFQUEUE table. Because the uda.fmb form in RMS automatically removes any child records on the UDA_VALUES table when the parent uda is removed, there will be a row inserted to the UDA_MFQUEUE table for each uda_value record associated with the deleted UDA as well. These will receive the lower sequence numbers so that these will be acted upon first in the message queue. They will look like the DELETE UDA_VALUES message detailed in the section above.
3. Message: A UDARef message is created for the parent UDA only. It is a flat, synchronous message containing the primary key with which the external systems can remove it from their systems.

Package Name: RMSMFM_UDA

Spec File Name: rmsmfm_udas.pls

Body File Name: rmsmfm_udab.pls

Package Specification – Global Variables

None

Function Level Description – ADDTOQ

```
Function: ADDTOQ
              (O_status             OUT     VARCHAR2,
               O_text               OUT     VARCHAR2,
               I_message_type       IN      UDA_MFQUEUE.MESSAGE_TYPE%TYPE,
               I_uda_id             IN      UDA.UDA_ID%TYPE,
               I_uda_value          IN      UDA_VALUES.UDA_VALUE%TYPE,
               I_display_taype      IN      UDA_MFQUEUE.DISPLAY_TYPE%TYPE,
               I_message            IN      CLOB
                          )
```

This procedure is called by the triggers and takes the message type, uda_id and uda_value if there is one and the message itself. It inserts a row into the UDA_MFQUEUE along with the passed in values and the next sequence number from the UDA_MFSEQUENCE, setting the status to ‘U’npublished. It returns error codes and strings.

Function Level Description – GETNXT

```
Procedure: GETNXT
              (O_status_code        OUT     VARCHAR2,
               O_error_msg          OUT     VARCHAR2,
               O_message_type       OUT     UDA_MFQUEUE.MESSAGE_TYPE%TYPE,
               O_message            OUT     CLOB,
               O_uda_id             OUT     UDA.UDA_ID%TYPE,
               O_uda_value          OUT     UDA_VALUES.UDA_VALUE%TYPE,
               O_display_type       OUT     UDA_MFQUEUE.DISPLAY_TYPE%TYPE
              )
```

This publicly exposed procedure is typically called by a RIB publication adaptor. This procedure’s parameters are well defined and arranged in a specific order. The message type is the RIB defined short message name; the message is the XML message; and the uda_id and uda_value are the keys for the message as pertains to the UDA family, not all of which will necessarily be populated for all message types. The status code is one of five values.

Trigger Impact

Create a trigger on the UDA and UDA_VALUES table to capture Inserts, Updates, and Deletes.

Trigger Name: EC_TABLE_UDA_AIUDR.TRG

Trigger File Name: ec_table_uda_aiudr.trg

Table: UDA

Inserts

- Sets action_type to ‘A’dd and message_type to ‘UDAHdrCre’.

Updates

- Sets action_type to ‘M’odify and message_type to ‘UDAHdrMod’.

Deletes

- Sets action_type to ‘D’elete and message_type to ‘UDAHdrDel’.

Trigger Name: EC_TABLE_UDV_AIUDR.TRG

Trigger File Name: ec_table_udv_aiudr.trg

Table: UDA_VALUES

Inserts

- Sets action_type to ‘A’dd and message_type to ‘UDAValCre’.

Updates

- Sets action_type to ‘M’odify and message_type to ‘UDAValMod’.

Deletes

- Sets action_type to ‘D’elete and message_type to ‘UDAValdel’.

DTD

Here are the filenames that correspond with each message type. Please consult RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
UDAHdrCre	UDA Create Message	UDADesc.dtd
UDAHdrMod	UDA Modify Message	UDADesc.dtd
UDAHdrDel	UDA Delete Message	UDARef.dtd
UDAValCre	UDA_Values Create Message	UDAValDesc.dtd
UDAValMod	UDA_Values Modify Message	UDAValDesc.dtd
UDAValDel	UDA_Values Delete Message	UDAValRef.dtd

Table Impact

TABLE	SELECT	INSERT	UPDATE	DELETE
UDA_MFQUEUE	Yes	Yes	No	No

Design Assumptions

- One of the primary assumptions in the current approach is that ease of code will outweigh performance considerations. It is hoped that the ‘trickle’ nature of the flow of data will decrease the need to dwell on performance issues and instead allow developers to code in the easiest and most straightforward manner.
- The adaptor is only set up to call stored procedures, not stored functions. Any public program then needs to be a procedure.

Vendor

Business Overview

RMS publishes vendor (also known as supplier) and vendor address (also known as supplier address) data messages to the RIB for the use of RWMS and other integration subsystems. Those applications are then able to keep their vendor data in sync with RMS.

Supplier data is a base foundation data element that every Oracle Retail system uses. RMS publishes exhaustive supplier data. The subscribing application filters out the data it needs.

As suppliers and addresses are added in RMS, the event capture trigger creates an xml message in the xml builder and adds the message to the SUPPLIER_MFQUEUE table. The supplier create message will consist of two parts - the header (sups table) and the detail (addr table) data. The number of addresses needed is determined by system options, invoice matching indicator and returns allowed indicator, with an order address type always being required. Once all the criteria are met for a valid create message, the messages will be combined and sent to the RIB. Messages for supplier and address modifications and deletions will be sent as they are created. An address modification can be sent without the supplier information.

The tables involved are sups and addr. Address records are children of suppliers. All address types of Returns (3), Order (4), and Invoice (5) are published.

Functionality Checklist

Description	RMS	RIB
RMS must publish vendor information		
Create new Publisher	X	X

Form Impact

None.

Business Object Records

None.

Package Impact

Description of Activities

<Create Supplier>

1. Prerequisites: Supplier does not already exist.
2. Activity Detail: Create Supplier inserts the supplier into the database.
3. Messages: VendorCre message is created but remains in queue until supplier is valid.

<ModifySupplier>

1. Prerequisites: Supplier exists and is valid
2. Activity Detail: Update the supplier record in the database.
3. Messages: VendorHdrMod message is created.

<AddAddress>

1. Prerequisites: Supplier exists and is valid
2. Activity Detail: Insert the address into the database.
3. Messages: VendorAddrCre message is created.

<ModifyAddress>

1. Prerequisites: Supplier exists and is valid. Address exists.
2. Activity Detail: Update the address in the database.
3. Messages: VendorAddrMod message is created.

<DeleteAddess>

1. Prerequisites: Supplier exists and is valid. Address exists.
2. Activity Detail: Delete the address from the database.
3. Messages: VendorAddrDel message is created.

<DeleteSupplier>

1. Prerequisites: Supplier exists and is valid.
2. Activity Detail: Delete any existing addresses for the supplier and the supplier from the database.
3. Messages: VendorDel message is created.

Package Name: RMSMFM_VENDOR

Spec File Name: rmsmfm_vendors.pls

Body File Name: rmsmfm_vendorb.pls

Package Specification – Global Variables

None

Function Level Description – ADDTOQ

```
Function: ADDTOQ
               (I_message_type       IN     VARCHAR2,
                I_supplier           IN     sups.supplier%TYPE,
                I_addr_seq_no        IN     addr.seq_no%TYPE,
                I_addr_type          IN     addr.addr_type%TYPE,
                I_ret_allow_ind      IN     VARCHAR2,
                I_invc_match_ind     IN     VARCHAR2,
                I_message            IN     CLOB,
                O_status             OUT    VARCHAR2,
                O_text               OUT    VARCHAR2)
```

This procedure is called by the triggers, and takes the message type, supplier, addr_seq_no, addr_type, ret_allow_ind, and invc_match_ind values and, the message itself. It inserts a row into the supplier message family queue along with the passed in values and the next sequence number from the supplier message family sequence, setting the status to unpublished. It returns error codes and strings.

Function Level Description – GETNXT

```
Procedure: GETNXT
              (O_status_code        OUT    VARCHAR2,
               O_error_msg          OUT    VARCHAR2,
               O_message_type       OUT    VARCHAR2,
               O_message            OUT    CLOB,
               O_supplier           OUT    sups.supplier%TYPE
               O_addr_seq_no        OUT    addr.seq_no%TYPE
               O_addr_type          OUT    addr.addr_type%TYPE
```

This publicly exposed procedure is called by a RIB publication adaptor. Its parameters are well defined and arranged in a specific order. The message type is the RIB defined short message name, the message is the xml message, and the family key(s) are the key for the message as pertains to the family, not all of which will necessarily be populated for all message types. The keys for supplier are supplier, addr_seq_no, and addr_type. Status code is one of 3 values, as shown in the following table.

The error text parameter contains application-generated information, such as the application's sequence number of the message that failed, and the Oracle or other error that occurred when the retrieval failed.

NO_MSG	'N'	No more messages to process
UNHANDLED_ERROR	'E'	Unclassified (fatal) Error
SUCCESS	'S'	Success

Function Level Description – CREATE_PREVIOUS (local)

This procedure determines if a supplier create already exists on the queue table for the same supplier and with a sequence number less than the current records sequence number.

Function Level Description – CLEAN_QUEUE (local)

This procedure cleans up the queue by eliminating modification messages. It is only called if CREATE_PREVIOUS returns true. For each address modification message type, it finds the previous address create message type. It then calls REPLACE_QUE_ADR to copy the modify message into the create message and calls DELETE_QUEUE_REC to delete the modify record. For each delete message type, it finds the previous corresponding create message type. It then calls DELETE_QUEUE_REC to delete the create message record. For each supplier modification message type, it finds the previous supplier create message type. It then calls REPLACE_QUE_SUP to copy the modify message into the create message and calls DELETE_QUEUE_REC to delete the modify record.

Function Level Description – CAN_CREATE (local)

This procedure determines if a complete hierarchical supplier message can be created from the current address and prior address messages in the queue for the same supplier. It checks to see if there is a type 3, 4, and 5 address already in the queue. If the ret_allow_ind is 'Y' and there is a type 3 address, then a ret_flag is set to true. If the invc_match_ind is 'Y' and there is a type 5 address, then a invc_flag is set to true. If all the flags are true, then it returns true because the complete hierarchical message can be created.

Function Level Description – MAKE_CREATE (local)

This procedure combines the current message and all previous messages with the same supplier in the queue table to create the complete hierarchical message. It first creates a new message with the VendorDesc document type. It then gets the supplier create message and adds it to the new message. The remainder of this procedure gets each of the addresses adds them to the new message. When it is finished creating the new message, it deletes all the records from the queue with a sequence number less than or equal to the current records sequence number. This new message is passed back to the bus.

Function Level Description – DELETE_QUEUE_REC (local)

This procedure deletes a specific record from the queue. It deletes based on the sequence number passed in.

Function Level Description – REPLACE_QUEUE_SUP (local)

This procedure replaces the message in the create supplier record with the message from the modify supplier record.

Function Level Description – REPLACE_QUEUE_ADR (local)

This procedure replaces the message in the create address record with the message from the modify address record.

Function Level Description – CHECK_STATUS (local)

This procedure raises an exception if the status code is set to Error. This should be called immediately after calling a procedure that sets the status code. Any procedure that calls CHECK_STATUS must have its own exception handling section.

Trigger Impact

Create a trigger on the SUPS and ADDR tables to capture Inserts, Updates, and Deletes. Triggers should only insert records onto the staging table.

Trigger Name: EC_TABLE_SUP_AIUDR.TRG

Trigger File Name: ec_table_sup_aiudr.trg

Table: SUPS

Description:

This trigger fires on insert, update, and delete. It captures the data in new. It then sets the event type and message type and calls the supplier_xml.build_supplier procedure. It calls supplier_xml.get_keys to get the key, returns allowed indicator, and invoice match indicator. The message is then inserted into the mfqueue table by calling rmsmfm_supplier.addtoq.

Inserts

- Set event type to ‘A’ and message type to VendorHdrCre.

Updates

- Set event type to ‘D’ and message type to VendorHdrMod.

Deletes

- Set event type to ‘D’ and message type to VendorDel.

Trigger Name: EC_TABLE_ADR_AIUDR.TRG

Trigger File Name: ec_table_adr_aiudr.trg

Table: ADDR

Description:

This trigger fires on insert, update, and delete. It captures the data in new. It then sets the event type and message type and calls the supplier_xml.build_address procedure. It calls supplier_xml.get_keys to get the key, returns allowed indicator, and invoice match indicator. The message is then inserted into the mfqueue table by calling rmsmfm_supplier.addtoq.

Inserts

- Set event type to 'A' and message type to VendorAddrCre.

Updates

- Set event type to 'D' and message type to VendorAddrMod.

Deletes

- Set event type to 'D' and message type to VendorAddrDel.

DTD

Here are the filenames that correspond with each message type. Please consult RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
VendorCre	Vendor Create	VendorDesc.dtd
VendorHdrMod	Vendor Header Modify	VendorHdrDesc.dtd
VendorDel	VendorDelete	VendorRef.dtd
VendorAddrCre	Vendor Address Create	VendorAddrDesc.dtd
VendorAddrMod	Vendor Address Modify	VendorAddrDesc.dtd
VendorAddrDel	Vendor Address Delete	VendorAddrRef.dtd

Table Impact

TABLE	SELECT	INSERT	UPDATE	DELETE
SUPS	Yes	No	No	No
ADDR	Yes	No	No	No
SUPPLIER_MFQUEUE	Yes	Yes	Yes	Yes
DUAL	Yes	No	No	No

Design Assumptions

- One of the primary assumptions in the current approach is that ease of code will outweigh performance considerations. It is hoped that the ‘trickle’ nature of the flow of data will decrease the need to dwell on performance issues and instead allow developers to code in the easiest and most straight forward manner.
- The adaptor is only setup to call stored procedures, not stored functions. Any public program then needs to be a procedure.

Warehouse

Business Overview

RMS publishes data about warehouses in messages to the Oracle Retail Integration Bus (RIB). Other applications that need to keep their locations synchronized with RMS subscribe to these messages.

RMS publishes event messages about all of its warehouses. For retailers that run RMS in a multi-channel environment, this is important because RMS warehouses are divided into those that hold inventory, called stockholding, and those that do not hold inventory, called non-stockholding. Stockholding locations can include virtual warehouses and brick and mortar stores. Actual physical warehouses are non-stockholding for RMS' purposes.

Those applications on the RIB that understand virtual locations can subscribe to all warehouse messages that RMS publishes. Those applications that do not have virtual location logic, that is they only understand a location as physical and stockholding, depend upon the RIB to transform RMS warehouse messages. Logic contained in the RIB ensures that these applications will not receive virtual warehouse data.

> **Note:** To determine if your implementation of RMS is set up to run multi-channel, look at the SYSTEM_OPTIONS table's multichannel_ind column for the value of 'Y' (Yes). If the 'N' (No) value is displayed, multichannel is not enabled.

These RIB messages are triggered by the insert, update, and delete of warehouses and warehouse addresses on the RMS WH table, and the ADDR table with the module 'WH'. The output message is in hierarchical structure, with warehouse information at the header level and the address information at the detail level. Because this is a low volume publisher, multi-threading capability is not supported. In addition, Oracle Retail only publishes the current state of the warehouse, not every change.

If multiple addresses are associated with a warehouse, this message is designed with the assumption that RWMS and other integration subsystems only subscribe to the primary address of the primary address type.

To facilitate the routing of data, the header level routing info (this is routing info outside the confines of the message) contains the name of 'loc_type' with value 'W'. Detail level routing info contains the name of 'primary_addr_type_ind' with value of 'Y' or 'N' and the name of 'primary_addr_ind' with value of 'Y' or 'N'. This allows the RIB to route the external finishers and their address to the correct applications.

For additional explanations of virtual locations and the multi-channel operation of RMS, see the chapter "Organization Hierarchy Batch" in volume 1 of this RMS Operations Guide. For an explanation of finishers, see the section 'Partner' in the chapter "Publication Design" in this volume of the RMS Operations Guide.

Functionality Checklist

Description	RMS	RIB
RMS must publish WH information		
Create new Publisher	X	X

Business Object Records

Create the following record types in the RMSFM_WH package specification:

```
TYPE WH_KEY_REC IS RECORD
       (wh                NUMBER,
       addr_key           NUMBER,
       wh_type            VARCHAR2(1),
       pricing_loc        NUMBER,
       pricing_loc_curr   VARCHAR2(3)
       );
```

Package Impact

Business Object ID

The business object id for warehouse publisher is wh, which uniquely identifies a warehouse for publishing.

Package Name: RMSMFM_WH

Spec File Name: rmsmfm_whs.pls

Body Ffile Name: rmsmfm_whb.pls

Package Specification – Global Variables

```
FAMILY     CONSTANT   RIB_SETTINGS.FAMILY%TYPE      := 'WH';
HDR_ADD    CONSTANT   VARCHAR2(15)                  := 'whcre';
HDR_UPD    CONSTANT   VARCHAR2(15)                  := 'whmod';
HDR_DEL    CONSTANT   VARCHAR2(15)                  := 'whdel';
DTL_ADD    CONSTANT   VARCHAR2(15)                  := 'whdtlcre';
DTL_UPD    CONSTANT   VARCHAR2(15)                  := 'whdtlmod';
DTL_DEL    CONSTANT   VARCHAR2(15)                  := 'whdtldel';
WHA_ADD    CONSTANT   VARCHAR2(15)                  := 'whaddcre';
WHA_UPD    CONSTANT   VARCHAR2(15)                  := 'whaddmod';
```

Function Level Description – ADDTOQ

```
Function: ADDTOQ
          (O_error_mesage           OUT     VARCHAR2,
           I_message_type           IN      VARCHAR2,
           I_wh_key_rec             IN      WH_KEY_REC,
           I_addr_publish_ind       IN      ADDR.PUBLISH_IND%TYPE)
```

This public function puts a warehouse message on WH_MFQUEUE for publishing to the RIB. It is called from both wh trigger and address trigger. The I_functional_keys contains wh and, optionally, addr_key.

There are some tasks relating to streamlining the queue clean up process that need to occur in ADDTOQ. The goal is to have at most one record on the queue for a given warehouse up until its initial publication.

- For header level insert messages (HDR_ADD), insert a record in the WH_PUB_INFO table. The published flag should be set to 'N'. The WH_TYPE column should have a value of 'P'hysical or 'V'irtual. Since this is a low volume business transaction, the warehouse publisher will not provide multi-threading capability. Therefore, thread value does NOT need to be calculated.
- For HDR_UPD, DTL_ADD, DTL_UPD, DTL_DEL messages, do NOT need to be added to the queue until the business object (that is, wh) has been initially published (WH_PUB_INFO.published = 'N'). This is because when the warehouse does get published for the first time (HDR_ADD), the current state of the header and details is queried and published to the RIB.
- For header level delete messages (HDR_DEL), if the business object (that is, wh) has NOT been initially published (WH_PUB_INFO.published = 'N'), delete the record in WH_PUB_INFO. Otherwise, delete every record in the queue for the business object, and add the delete record to the queue.
- If the business object has been initially published (WH_PUB_INFO.published = 'Y'), for detail level messages deletes (DTL_DEL), Oracle Retail only needs one (the most recent) record per detail in the WH_MFQUEUE. Delete any previous records that exist on the WH_MFQUEUE for the record that has been passed.
- If the business object has been initially published (WH_PUB_INFO.published = 'Y'), for detail level messages updates (DTL_UPD), Oracle Retail only needs one DTL_UPD (the most recent) record per detail in the WH_MFQUEUE. Delete any previous DTL_UPD records that exist on the WH_MFQUEUE for the record that has been passed. Oracle Retail does not want to delete any detail inserts that exist on the queue for the detail, the system needs to ensure subscribers are not passed a detail modification message for a detail that they do not yet have.
- For all message types except header level inserts (HDR_ADD), insert a record into the WH_MFQUEUE. One exception is that if the publish_ind on the detail record table (ADDR) is 'N', do not add the DTL_DEL message to the queue. That is because if the detail line has never been published before, subscribers will not need to delete the detail line that they do not yet have.

Function Level Description – GETNXT

```
Procedure: GETNXT
              (O_status_code        OUT    VARCHAR2,
               O_error_msg          OUT    VARCHAR2,
               O_message_type       OUT    VARCHAR2,
               O_message            OUT    RIB_OBJECT,
               O_bus_obj_id         OUT    RIB_BUSOBJID_TBL,
               O_routing_info       OUT    RIB_ROUTINGINFO_TBL,
               I_num_threads        IN     NUMBER DEFAULT 1,
               I_thread_val         IN     NUMBER DEFAULT 1);
```

This public procedure is called from the RIB to get the next messages. It performs a cursor loop on the unpublished records on the WH_MFQUEUE table (PUB_STATUS = 'U'). It should only need to execute one loop iteration in most cases. For each record retrieved, GETNXT gets the following:

1. A lock of the queue table for the current business object (that is, wh). The lock is obtained by calling the function LOCK_THE_BLOCK. If there are any records on the queue for the current business object that are already locked, the current message is skipped and picked up again in the next loop iteration.
2. A check for records on the queue with a status of 'H'ospital. If there are any such records for the current business object, GETNXT raises an exception to send the current message to the Hospital.
3. Get the published indicator from the WH_PUB_INFO table.
4. Call PROCESS_QUEUE_RECORD with the current business object.

The loop will need to execute more than one iteration for the following cases:

1. When a header delete message exists on the queue for a business object that has not been initially published. In this case, simply remove the header delete message from the queue and loop again.
2. The queue is locked for the current business object. This can occur because ADDTOQ, which is called from the triggers, deletes from the queue table for DTL_UPD, DTL_DEL, and HDR_DEL messages.

The information from the WH_MFQUEUE and WH_PUB_INFO table is passed to PROCESS_QUEUE_RECORD. PROCESS_QUEUE_RECORD will build the Oracle Object message to pass back to the RIB. If PROCESS_QUEUE_RECORD does not run successfully, GETNXT raises an exception.

If any exception is raised in GETNXT, including the exception raised by an unsuccessful call to PROCESS_QUEUE_RECORD, HANDLE_ERRORS is called.

Function Level Description – PUB_RETRY

This public procedure performs the same tasks as GETNXT except that it only loops for a specific row in the WH_MFQUEUE table. The record on WH_MFQUEUE must match the passed in sequence number (contained in the ROUTING_INFO).

Function Level Description – PROCESS_QUEUE_RECORD (local)

This private function controls the building of Oracle Objects (DESC or REF) given the business transaction's key values and a message type. It contains all of the shared processing between GETNXT and PUB_RETRY. (Note: message_type of HDR_ADD can potentially be changed to a DTL_ADD in PROCESS_QUEUE_RECORD.)

If the message type is a header delete (HDR_DEL)

- I_hdr_published should never be 'N' (not published), because in that case the HDR_DEL message would never be added to the queue based on ADDTOQ processing.
- Call BUILD_HEADER_OBJECT to build the REF Oracle Object to publish to the RIB.
- Delete both WH_MFQUEUE and WH_PUB_INFO for the business object.

Otherwise, if check I_hdr_published is either 'N' (not published) or 'I' (in progress):

- If I_hdr_published is 'N', the message type is a header create (HDR_ADD). If I_hdr_published is 'I', change the message type from a header create (HDR_ADD) to a detail add (DTL_ADD), because this is the situation where some of the details are published but not all due to MAX_DETAILS_TO_PUBLISH. Thus, a header create message for the current business object should have already been published.
- Call MAKE_CREATE to build the DESC Oracle Object to publish to the RIB. This will also take care of any WH_MFQUEUE deletes, updating WH_PUB_INFO.PUBLISHED to 'Y' or 'I', and bulk updating addr.publish_ind to 'Y' for those detail rows that have been published.

Otherwise,

If the record from WH_QUEUE table is a header update (HDR_UPD)

- Call BUILD_HEADER_OBJECT to build the DESC Oracle Object to publish to the RIB.
- Update WH_PUB_INFO to published = 'Y'.
- Delete the record from the WH_MFQUEUE table.

If the record from WH_QUEUE table is a detail add or update (DTL_ADD, DTL_UPD)

- Call BUILD_DETAIL_CHANGE_OBJECTS to build the DESC Oracle Object to publish to the RIB. This will also take care of any WH_MFQUEUE deletes and updates of publish_ind on ADDR.

If the record from WH_QUEUE table is a detail delete (DTL_DEL)

- Call BUILD_DETAIL_DELETE_OBJECTS to build the REF Oracle Object to publish to the RIB. This will also take care of any WH_MFQUEUE deletes.

Function Level Description – DELETE_QUEUE_REC (local)

This private function deletes a record in WH_MFQUEUE table given the row id.

Function Level Description – MAKE_CREATE (local)

```
Procedure: MAKE_CREATE
              (O_error_msg          OUT           VARCHAR2,
               O_message            IN  OUT       nocopy RIB_OBJECT,
               O_routing_info       IN  OUT       nocopy RIB_ROUTINGINFO_TBL,
               I_wh_key_rec         IN            WH_KEY_REC,
               I_rowid              IN            ROWID)
```

This private function is used to create the Oracle Object for the initial publication of a business transaction. I_business_object contains the warehouse header key values (wh). I_rowid is the rowid of the wh_mfqueue row fetched from GETNXT.

- Call BUILD_HEADER_OBJECT to get a header level Oracle Object.
- Call BUILD_DETAIL_OBJECTS to get a table of detail level Oracle objects and a table of WH_MFQUEUE rowids to delete with and a table of ADDR rowids to update publish_ind with.
- Update WH_PUB_INFO.published to ‘Y’ or ‘I’ depending on if all details are published.
- Delete records from the WH_MFQUEUE for all rowids returned by BUILD_DETAIL_OBJECTS. Deletes are done by rowids instead of business transaction keys to ensure that noting is deleted off the queue that has not been published.
- If the entire business transaction was added to the Oracle Object, also delete the WH_MFQUEUE record that was picked up by GETNXT. If the entire business transaction was NOT published, Oracle Retail needs to leave something on the WH_MFQUEUE to ensure that the rest of it is picked up by the next call to GETNXT.
- Update ADDR.publish_ind to ‘Y’ by addr rowids returned from BUILD_DETAIL_OBJECTS.
- The header and detail level Oracle Objects are combined and returned.

Function Level Description – BUILD_HEADER_OBJECT (local)

```
Procedure: BUILD_HEADER_OBJECT
              (O_error_msg          OUT           VARCHAR2,
               O_routing_info       IN  OUT       nocopy RIB_ROUTINGINFO_TBL,
               O_rib_whdesc_rec     OUT           RIB_WH_DESC,
               I_wh_key_rec         IN            WH_KEY_REC)
```

This private function accepts warehouse header key values (wh), builds and returns a header level DESC Oracle Object.

Function Level Description – BUILD_HEADER_OBJECT (local)

This overloaded private function accepts warehouse header key value (wh), builds and returns a header level REF Oracle Object.

Function Level Description – BUILD_DETAIL_OBJECTS (local)

The private function is responsible for building detail level DESC Oracle Objects. It builds as many detail Oracle Object as it can given the passed in message type and business object keys (wh).

Call API_LIBRARY.GET_RIB_SETTINGS to get the MAX_DETAILS_TO_PUBLISH for the warehouse family.

If the function is being called from MAKE_CREATE (I_message_type would be NULL):

Select any unpublished ADDR detail records for the business transaction (based on publish_ind on ADDR). Create Oracle Objects for details that are selected by calling BUILD_SINGLE_DETAIL.

- Ensure that no more than MAX_DETAILS_TO_PUBLISH records are put into Oracle Objects.
- Return a table of wh_mfqueue rowids for each message that is placed into the Oracle Object.
- Return a table of addr rowids for each detail that is placed into the Oracle Object.

If the function is **not** being called from MAKE_CREATE (I_message type will NOT be NULL):

Select any details on the WH_MFQUEUE that are for the same business object and for the same message type. Create Oracle Objects for details that are selected by calling BUILD_SINGLE_DETAIL.

- Ensure that no more than MAX_DETAILS_TO_PUBLISH records are put into Oracle Objects.
- If the message type is a detail create (DTL_ADD), as the detail info is placed into the Oracle Object, ensure that the corresponding addr rowid is added to addr rowids table for return. These rowids are used to update ADDR.publish_ind to 'Y'.
- Return a table of wh_mfqueue rowids for each message that is placed into the Oracle Object.

A concern here is making sure that deletions from the queue table do not occur for information that has not been published. For this reason, deletes are done by ROWID. Everything should be in the same cursor, this should ensure that the message published matches the deletes performed from the WH_MFQUEUE table regardless of trigger execution during GETNXT calls.

Function Level Description – BUILD_SINGLE_DETAIL (local)

This private function takes in an address record and builds a detail level Oracle Object. Also find out if the address is the primary address of the primary address type and set the DESC Oracle Object accordingly.

Function Level Description – BUILD_DETAIL_CHANGE_OBJECTS (local)

This private function builds a DESC Oracle Object to publish to the RIB for detail create and detail update messages (DTL_ADD, DTL_UPD). I_business_obj contains the header level key values (wh).

- Call BUILD_HEADER_OBJECT to build the header level DESC Oracle Object.
- Call BUILD_DETAIL_OBJECTS to build the detail level DESC Oracle Objects.
- Bulk update addr.publish_ind to 'Y' for the addr rowids returned from BUILD_DETAIL_OBJECTS.

Bulk delete from wh_mfqueue for the wh_mfqueue rowids returned from BUILD_DETAIL_OBJECTS.

Function Level Description – BUILD_DETAIL_DELETE_OBJECTS (local)

This private function builds a REF Oracle Object to publish to the RIB for detail delete messages (DTL_DEL). I_business_obj contains the header level key values (wh).

- Call API_LIBRARY.GET_RIB_SETTINGS to get the MAX_DETAILS_TO_PUBLISH for the warehouse family.
- Call BUILD_HEADER_OBJECT to build the REF Oracle Object to publish to the RIB.
- Perform a cursor for loop on WH_MFQUEUE and build as many detail REF Oracle Objects as possible without exceeding the MAX_DETAILS_TO_PUBLISH.
- For each detail, also find out if the address is a primary address of the primary address type, and set the REF Oracle Objects accordingly.
- Bulk delete from wh_mfqueue for the wh_mfqueue rowids queried.

Function Level Description – LOCK_THE_BLOCK (local)

This private function locks all queue records for the current business object (wh). This is to ensure that GETNXT and PUB_RETRY do not wait on any business processes that currently have the queue table locked and have not committed. This can occur because ADDTOQ, which is called from the triggers, deletes from the queue table for DTL_UPD, DTL_DEL, and HDR_DEL messages.

Function Level Description – HANDLE_ERRORS (local)

This private procedure is called from GETNXT and PUB_RETRY when an exception is raised. I_seq_no is the sequence number of the driving WH_MFQUEUE record. I_function_keys contains detail level key values (wh, addr_key).

If the error is a non-fatal error, HANDLE_ERRORS passes the sequence number of the driving WH_MFQUEUE record back to the RIB in the ROUTING_INFO. It sends back a status of 'H'ospital to the RIB as well. It then updates the status of the queue record to 'H'ospital, so that it will not get picked up again by the driving cursor in GETNXT.

If the error is a fatal error, a status of 'E'rror is returned to the RIB. The error is considered non-fatal if no DML has occurred yet. Whenever DML has occurred, then the global variable LP_error_status is flipped from 'H'ospital to 'E'rror.

Trigger Impact

Create a trigger on the WH and ADDR table to capture Inserts, Updates, and Deletes.

Trigger Name: EC_TABLE_WH_AIUDR.TRG (new)

Trigger File Name: ec_table_wh_aiudr.trg (new)

Table: WH

Create a trigger on the WH table to capture Inserts, Updates, and Deletes.

Inserts

- Send the header level warehouse info to the ADDTOQ procedure in the MFM with the message type RMSMFM_WH.HDR_ADD.

Updates

- Send the header level warehouse info to the ADDTOQ procedure in the MFM with the message type RMSMFM_WH.HDR_UPD.

Deletes

- Send the header level info to the ADDTOQ procedure in the MFM with the message type RMSMFM_WH.HDR_DEL.

In all these cases, build the function keys for ADDTOQ with warehouse.

Trigger Name: EC_TABLE_ADR_AIUDR.TRG (mod)

Trigger File Name: ec_table_adr_aiudr.trg (mod)

Table: ADDR

Modify trigger on the ADDR table to capture Inserts, Updates, and Deletes of module type ‘WH’ (currently it only handles supplier).

Inserts

- Send the detail level addr info to the ADDTOQ procedure in the MFM with the message type RMSMFM_WH.DTL_ADD.

Updates

- Send the detail level addr info to the ADDTOQ procedure in the MFM with the message type RMSMFM_WH.DTL_UPD.

Deletes

- Send the detail level info to the ADDTOQ procedure in the MFM with the message type RMSMFM_WH.DTL_DEL.

In all these cases, build the function keys for ADDTOQ with KEY_VALUE_1 and KEY_VALUE_2, which represent wh and addr_key respectively.

DTD

Here are the filenames that correspond with each message type. Please consult RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
WHCre	WH Create Message	WHDesc.dtd
WHMod	WH Modify Message	WHDesc.dtd
WHDel	WH Delete Message	WHRef.dtd
WHDtlCre	WH Detail Create Message	WHDesc.dtd
WHDtlMod	WH Detail Modify Message	WHDesc.dtd
WHDtlDel	WH Detail Delete Message	WHRef.dtd
WHAddCre	WH Address Create	WHAddrDesc.dtd
WHAddMod	WH Address Modify	WHAddrDesc.dtd

Table Impact

TABLE	SELECT	INSERT	UPDATE	DELETE
WH_MFQUEUE	Yes	Yes	Yes	Yes
WH_PUB_INFO	Yes	Yes	Yes	Yes
WH	Yes	No	No	No
ADDR	Yes	No	Yes	No
ADD_TYPE_MODULE	Yes	No	No	No

Design Assumptions

It is not possible for a detail trigger to accurately know the status of a header table. In order for the detail triggers to accurately know when to add a message to the queue, RMS should not allow approval of a business object while detail modifications are being made.

It is not possible for a header trigger or a detail trigger to know the status of anything modified by GETNXT. If a header trigger or detail trigger is trying to delete queue records that GETNXT currently has locked, it will have to wait until GETNXT is finished and removes the lock. It is assumed that this time is fairly short (at most 2-3 seconds.) It is also assumed that this will occur rarely, as it involves updating/deleting detail records on a business object that has already been approved. This also has to occur at the same time GETNXT is processing the current business object.

Push off all DML statements as late as possible. Once DML statements have taken place, any error becomes a fatal error rather than a hospital error.

Work Orders In

Purchase orders

Business Overview

A work order provides direction to a warehouse management system (such as RWMS) about work that needs to be completed on items contained in a recent purchase order. RMS publishes work orders soon after it publishes the purchase order itself. This is referred to as a 'work order in' message. This message is not to be confused with a 'work order out' message, which pertains to transfers. For information on the Work Orders Out publication, see the section 'Work Orders Out' in the chapter "Publication Design" in this volume of the RMS Operations Guide.

Work order publication consists of a message containing attributes from the WO_DETAIL table plus the order number from the WO_HEAD table. One message is created each time a WO_DETAIL record is created, modified, or deleted. The primary key for the WO_DETAIL consists of the work order ID, warehouse, item, location, and sequence number. Thus, one work order can have multiple Work Order Create messages. When a WO_DETAIL record is created or modified, the message contains a full snapshot of the WO_DETAIL record. When a WO_DETAIL record is deleted, the message contains a partial snapshot of the WO_DETAIL record. Messages are retrieved from the message queue in the order they were created.

Work orders attached to purchase orders will have their messages published after the order has been published. Work orders attached to previously published approved orders will have their messages published immediately.

Work orders are defined at the physical location level. The message family manager will send the warehouse at which the work order will be done. This is used by the RIB publication adaptor for routing messages to the appropriate warehouse.

Functionality Checklist

Description	RMS	RIB
RMS must publish work orders information		
Create new Publisher	X	X

Form Impact

None.

Business Object Records

Create the following table types in the RMSMFM_WOIN package:

TYPE rowid_TBL is table of ROWID INDEX BY BINARY_INTEGER;

Package Impact

Create

1. Prerequisites: An order has been distributed by item and location.
2. Activity Detail: A work order is ready to be published as soon as the order it is attached to has been published. An initial publication message is made.
3. Messages: A “Work Order Create” message is queued. This message contains a snapshot of the attributes on the WO_DETAIL table.

Modify

1. Prerequisites: Work order has been created.
2. Activity Detail: The user is allowed to change attributes of the work order detail record. These changes are of interest to other systems and so this activity results in the publication of a message. Work orders attached to purchase orders will have their messages published after the order has been published. Work orders attached to previously published approved orders will have their messages published immediately.
3. Messages: Any modifications to a work order detail record will cause a “Work Order Modify” message to be queued. This message contains the same attributes as the “Work Order Create” message.

Delete

1. Prerequisites: Work order has been created.
2. Activity Detail: Deleting a work order detail record removes it from the system. External systems are notified by a published message.
3. Messages: When a work order detail record is deleted a “Work Order Delete” message is queued. The message contains a partial snapshot of the WO_DETAIL table.

Package Name: RMSMFM_WOIN

Spec File Name: rmsmfm_woins.pls

Body File Name: rmsmfm_woinb.pls

Package Specification – Global Variables

```
FAMILY      VARCHAR2(64)       'woin';
WO_ADD      CONSTANT           VARCHAR2(20)          'InBdWOCre';
WO_UPD      CONSTANT           VARCHAR2(20)          'InBdWOMod';
WO_DEL      CONSTANT           VARCHAR2(20)          'InBdWODel';
```

Function Level Description – ADDTOQ

```
Function: ADDTOQ
              (O_error_msg          OUT          VARCHAR2,
               I_queue_rec          IN           WOIN_MFQUEUE%ROWTYPE,
               I_publish_ind        IN           WO_DETAIL.PUBLISH_IND%TYPE)
```

This procedure is called by EC_TABLE_WDL_AIUDR, and takes a record type variable that consists of columns from the WO_DETAIL table and message type. It inserts a row into the message family queue WOIN_MFQUEUE along with the passed in values and the next sequence number from the message family sequence, and sets the status to unpublished. It returns a status code of API_CODES.SUCCESS if successful, API_CODES.UNHANDLED_ERROR if not.

Function Level Description – GETNXT

```
Procedure: GETNXT
              (O_status_code        OUT          VARCHAR2,
               O_error_msg          OU           VARCHAR2,
               O_message_type       OU           VARCHAR2,
               O_message            OU           RIB_OBJECT,
               O_bus_obj_id         OU           RIB_BUSOBJID_TBL,
               O_routing_info       OU           RIB_ROUTINGINFO_TBL,
               I_num_threads        IN           NUMBER DEFAULT 1,
               I_thread_val         IN           NUMBER DEFAULT 1)
```

This publicly exposed procedure is typically called by a RIB publication adaptor. Its parameters are well defined and arranged in a specific order. The message type is the RIB defined short message name. Status code is one of five values. These codes are defined in the RIB_CODES package.

The error text parameter contains application-generated information, such as the application's sequence number of the message that failed, and the Oracle or other error that occurred when the retrieval failed.

Function Level Description – PUB_RETRY

```
Procedure: PUB_RETRY
              (O_status_code        OUT                  VARCHAR2,
               O_error_msg          OUT                  VARCHAR2,
               O_message_type       IN  OUT              VARCHAR2,
               O_message            OUT                  RIB_OBJECT,
               O_bus_obj_id         IN  OUT              RIB_BUSOBJID_TBL,
               O_routing_info       IN  OUT              RIB_ROUTINGINFO_TBL,
               I_REF_OBJECT         IN                   RIB_OBJECT);
```

Same as GETNXT except:

It only loops for a specific row in the WOIN_MFQUEUE table. The record on WOIN_MFQUEUE must match the passed in sequence number (contained in the ROUTING_INFO).

Function Level Description – PROCESS_QUEUE_RECORD (local)

This function controls the building of Oracle Objects given the business transaction's key values and a message type. It contains all of the shared processing between GETNXT and PUB_RETRY.

If the record from WOIN_QUEUE table is an insert or update (WO_ADD, WO_UPD):

- Build the header object that contains work order id and order number
- Call BUILD_DETAIL_OBJECTS to build the Oracle Object to publish to the RIB.

If the record from WOIN_QUEUE table is a delete (WO_DEL):

- Build the header object that contains work order id and order number.
- Call BUILD_DETAIL_DELETE_OBJECTS to build the Oracle Object to publish to the RIB.

Function Level Description – BUILD_DETAIL_OBJECTS (local)

The function is responsible for building detail level Oracle Objects. It builds as many detail Oracle Object as it can given the passed in message type and business object key (work order ID).

Select any details on the WOIN_MFQUEUE that are for the same work order id and for the same message type.

- WOIN_MFQUEUE records that contain information being published will be deleted.
- Each location represented in the published message will be added to the ROUTING_INFO object.
- No more than the MAX_DETAILS_TO_PUBLISH number of records are put into Oracle Objects.

To avoid deleting information from the queue table that has not been published, deletes are accomplished using ROWIDs. All information should be fetched using the same cursor, this should ensure that the published message matches the deletes from the WOIN_MFQUEUE table regardless of trigger execution during GETNXT calls.

Function Level Description – BUILD_DETAIL_DELETE_OBJECTS (local)

Perform a cursor for loop on WOIN_MFQUEUE and build as many detail ref Oracle Objects as possible without exceeding the MAX_DETAILS_TO_PUBLISH.

Perform any BULK DML statements for deletion from WOIN_MFQUEUE.

Each location represented in the published message will be added to the ROUTING_INFO object.

Function Level Description – LOCK_THE_BLOCK (local)

This function locks all queue records for the current business object. This is to ensure that GETNXT does not wait on any business processes that currently have the queue table locked and have not committed. This can occur because ADDTOQ, which is called from the triggers, deletes from the queue table for WO_DEL messages.

Function Level Description – ROUTING_INFO_ADD (local)

This function is called from within the BUILD_DETAIL_OBJECTS and BUILD_DETAIL_DELETE_OBJECTS. It will add the location from the message to the routing_info whenever a new location is added to the object being published.

Function Level Description – HANDLE_ERRORS (local)

HANDLE_ERRORS is called from GETNXT and PUB_RETRY when an exception is raised.

If the error is a non-fatal error, GETNXT passes the sequence number of the driving WOIN_MFQUEUE record back to the RIB in the ROUTING_INFO. It sends back a status of 'H'ospital to the RIB as well. It then updates the status of the queue record to 'H'ospital, so that it will not get picked up again by the driving cursor in GETNXT.

If the error is a fatal error, a status of 'E'rror is returned to the RIB.

The error is considered non-fatal if no DML has occurred yet. Whenever DML has occurred, then the global variable LP_error_status is flipped from 'H'ospital to 'E'rror.

Trigger Impact

Create a trigger on the WO_DETAIL to capture Inserts, Updates, and Deletes.

Trigger Name: EC_TABLE_WDL_AIUDR.TRG

Trigger File Name: ec_table_wdl_aiudr.trg

Table: WO_DETAIL

This trigger will capture inserts/updates/deletes to the WO_DETAIL table and write data into the WOIN_MFQUEUE message queue.

Inserts

- Send the header level info to the ADDTOQ procedure in the MFM with the message type RMSMFM_WOIN.WO_ADD.

Updates

- Send the header level info to the ADDTOQ procedure in the MFM with the message type RMSMFM_ WOIN.WO_UPD.

Deletes

- Send the header level info to the ADDTOQ procedure in the MFM with the message type RMSMFM_ WOIN.WO_DEL.

DTD

Here are the filenames that correspond with each message type. Please consult RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
InBdWOCre	Work Order Create Message	WODesc.dtd
InBdWOMod	Work Order Modify Message	WODesc.dtd
InBdWODel	Work Order Delete Message	WORef.dtd

Table Impact

TABLE	SELECT	INSERT	UPDATE	DELETE
WOIN_MFQUEUE	Yes	Yes	Yes	Yes
WO_DETAIL	Yes	No	Yes	No

Design Assumptions

- One of the primary assumptions in the current approach is that ease of code will outweigh performance considerations. It is hoped that the ‘trickle’ nature of the flow of data will decrease the need to dwell on performance issues and instead allow developers to code in the easiest and most straightforward manner.
- The adaptor is only setup to call stored procedures, not stored functions. Any public program then needs to be a procedure.

Work Orders Out

Transfers

Business Overview

This publication API facilitates the transmission of outbound work orders (OWO) from RMS to external systems. Only transfers that pass through a finisher before reaching the final location may be associated with work orders. The work orders are published upon approval of their corresponding transfers. The work order provides instructions for one or more of the following tasks to be completed at the finisher location:

- Perform some activity on an item, such as monogramming.
- Transform an item from one thing into another, such as dyeing a white t-shirt black.
- Combine bulk items into a pack or break down a pack into its component items.

Outbound work orders have their own message family because they cannot be bundled with transfer messages. This is because multi-legged transfers can be routed to either internal finishers (held as virtual warehouses) or external finishers (held as partners). Transfers to and from an internal finisher involve at least one book transfer. Because external systems may be unaware of virtual warehouses, book transfers are not communicated to external systems.

Outbound work order data is only published upon approval of the associated transfer. As such, *all* work order activity, transformation and packing data are contained in the same message. Because RMS does not allow users to modify work order activity, transformation or packing information for an approved transfer, detail-level messages of any type (create, delete, update) are never published. Outbound work order delete messages are published when the second leg of a multi-legged transfer is unapproved. This can be accomplished through the un-approval of an entire multi-legged transfer or the un-approval of the second leg only. A two-leg transfer that has had the first leg shipped can be set back to 'In Progress' status in order to make changes to the work order activities and the final location. When action has occurred, only the second leg is really set back to in progress. The first leg remains in shipped status.

Functionality Checklist

Description	RMS	RIB
RMS must publish Work Orders OUT information		
Create new Publisher	X	X

Form Impact

None.

Business Object Records

None.

Package Impact

Approve

1. Prerequisites: A multi-legged transfer must be approved and have work order details for each transfer detail.
2. Activity Detail: Approving a transfer changes the status of the transfer. This change in status signifies the first time systems external to RMS will have an interest in the existence of the transfer and work order.
3. Messages: When a transfer with finishing is approved, an “outbdwocre” message is inserted into the queue. The family manager creates a hierarchical message containing a full snapshot of the transfer work order details at the time the message is published.

Delete

1. Prerequisites: The associated transfer has finishing and is being deleted.
2. Activity Detail: Deleting a transfer removes it, and the associated work order from the system. External systems are notified by a published Delete message that contains the number of the transfer work order to be deleted.
3. Message: When a transfer with finishing is deleted, an “outbdwodel”, which is a flat notification message, is queued.

Unapprove

1. Prerequisites: A transfer with finishing is unapproved
2. Activity Detail: Unapproving a transfer changes the status to worksheet, which allows modification to the work order, transformation, packing, and item details. External systems are notified by a published Delete message that contains the number of the transfer work order to be deleted.
3. Messages: Unapproving a transfer queues an “outbdwounaprv” request. This results in an “outbdwodel” message being published, which is a flat notification message.

Package Name: RMSMFM_WOOUT

Spec File Name: rmsmfm_woouts.pls

Body File Name: rmsmfm_wooutb.pls

Package Specification – Global Variables

None.

Function Level Description – ADDTOQ

```
Function: ADDTOQ
              (O_error_mesage       OUT    VARCHAR2,
               I_message_type       IN     VARCHAR2,
               I_tsf_wo_id          IN     tsf_wo_head.tsf_wo_id%TYPE)
```

There are some tasks relating to streamlining the queue clean up process that need to occur in ADDTOQ. The goal is to have at most one record on the queue for business transactions up until their initial publication.

- For header level insert messages (HDR_ADD), insert a record in the WOOUT_PUB_INFO table. The work order number passed to the function should be inserted into the TSF_WO_ID column, and the published column should contain 'N'.
- If the business transaction has not been approved (woout_pub_info.publish_ind = 'N') and the triggering message is one of HDR_DEL and HDR_ANAPPRV, the record is not added to queue.

Function Level Description – GETNXT

```
Procedure: GETNXT
              (O_status_code        OUT    VARCHAR2,
               O_error_msg          OUT    VARCHAR2,
               O_message_type       OUT    VARCHAR2,
               O_message            OUT    RIB_OBJECT,
               O_bus_obj_id         OUT    RIB_BUSOBJID_TBL,
               O_routing_info       OUT    RIB_ROUTINGINFO_TBL,
               I_num_threads        IN     NUMBER DEFAULT 1,
               I_thread_val         IN     NUMBER DEFAULT 1)
```

This function fetches a record from the WOOUT_MFQUEUE table. Fetch the record that has the lowest sequence number among queue records that have a pub_status of 'U' and a thread_no that matches the I_thread_val.

Call the LOCK_THE_BLOCK function. If it determines that WOOUT_MFQUEUE is locked for a particular work order, set the sequence limit local variable to the current sequence number. This will prevent the GETNXT function from attempting to lock and process the same work order message over and over again in the loop.

Query the WOOUT_MFQUEUE table to determine if any records for the work order have been sent to the error hospital. If so, produce the 'SEND_TO_HOSP' error message and halt processing.

> **Note:** The only scenario in which a hospitalized record with the same tsf_wo_id as the message currently being processed would be found is if the initial HDR_ADD message had been hospitalized and a subsequent HDR_DEL or HDR_UNAPRV was being processed.

Call the PROCESS_QUEUE_RECORD function. If the break loop indicator returned from process_queue_record is TRUE, set the O_message_type output parameter to the message type fetched from the queue and return TRUE.

If the message type is null, set the status code output parameter to API_CODES.NO_MSG. Otherwise, set it to API_CODES.NEW_MSG and set the O_bus_obj_id parameter to RIB_BUSOBJID_TBL(L_tsf_wo_id).

Function Level Description – PUB_RETRY

```
Procedure: PUB_RETRY
              (O_status_code        OUT          VARCHAR2,
               O_error_msg          OUT          VARCHAR2,
               O_message_type       IN OUT       VARCHAR2,
               O_message            OUT          RIB_OBJECT,
               O_bus_obj_id         IN OUT       RIB_BUSOBJID_TBL,
               O_routing_info       IN OUT       RIB_ROUTINGINFO_TBL,
               I_REF_OBJECT         IN           RIB_OBJECT);
```

This procedure is called from the RIB for woout_mfqueue.seq_no's that have been placed in the RIB's error hospital. It functions similarly to GETNEXT, except that it only fetches the record from WOOUT_MFQUEUE that contains the sequence number passed by the RIB.

If the message's tsf_wo_id is null, raise an API_CODES.NO_MSG error. Call LOCK_THE_BLOCK. If the queue record is locked by another process, set the status code to API_CODES.HOSPITAL. If the queue record is not locked by another process, call PROCESS_QUEUE_RECORD. If the message returned from process_queue_record is null, raise the API_CODES.NO_MSG error. Otherwise, if the message object is populated, populate the business object table with the current work order number.

Function Level Description – PROCESS_QUEUE_RECORD (local)

This function controls the building of Oracle Objects given the business transaction's key values and a message type. It contains all of the shared processing between GETNXT and PUB_RETRY.

Check to see if the business object is being published for the first time. If the published_ind on the pub_info table is 'N', the business object is being published for the first time.

This function will set the O_break_loop parameter to FALSE in the following scenarios;

1. Processing a HDR_UNAPRV message for a work order that has a woout_pub_info.published of 'N'.
2. Processing a HDR_DEL message for a work order that has a woout_pub_info.published of 'N'.

The loop is not broken in these scenarios because they do not necessitate the publication of a message. Therefore, processing should continue so a message can be outputted.

If the message type is HDR_DEL and the work order has been published the function creates a work order ref object, and routing info object.

> **Note:** WO out routing info requires a 'to_loc' string and value.

If the message type is a HDR_UNAPRV and the work order has been published create a work order ref object and a routing info object. For all records associated with the work order on the tsf_wo_detail, tsf_xform_detail and tsf_packing tables, set the publish_ind to 'N'.

> **Note:** A published value of 'I'n progress indicates that the work order was being published but it had more detail records than allowed for a single message. The maximum detail per message value can be found on the rib_settings table for each message family.

If the published indicator is 'N', set the message type to HDR_ADD and call the MAKE_CREATE function.

If the published indicator is 'I', set the message type to DTL_ADD and call the MAKE_CREATE function.

Function Level Description – MAKE_CREATE (local)

This function will first call the BUILD_HEADER_OBJECT function.

- It will then call the BUILD_DETAIL_OBJECTS function and update the woout_pub_info column.
- It will also update the published_ind columns on TSF_WO_DETAIL, TSF_XFORM_DETAIL and TSF_PACKING.

Function Level Description – BUILD_HEADER_OBJECT (local)

This function fetches the transfer number and transfer parent number associated with the passed in work order number. Then, call the constructor for the rib_wooutdesc_rec, passing in the work order number, transfer number, and transfer parent number. Finally, it builds the routing info object.

Function Level Description – BUILD_DETAIL_OBJECTS (local)

The function is responsible for building detail level Oracle Objects. It builds as many detail Oracle Object as it can given the passed in message type and business object keys.

If the function is being called from MAKE_CREATE:

- Select any unpublished detail records from the business transaction (tsf_wo_detail, tsf_xfrom_detail, tsf_packing).
  - Ensure that WOOUT_MFQUEUE is deleted from as needed. If there is more than one WOOUT_MFQUEUE record for a detail level record, make sure they all get deleted. Current state should be considered, not every change.
  - Ensure that ROUTING_INFO is constructed if routing information is stored at the detail level in the business transaction.
  - Ensure that no more than MAX_DETAILS_TO_PUBLISH records are put into Oracle Objects.
  - Ensure that the detail records being added to the object have not already been published. This can happen if GETNXT was previously called for the current business object, and the MAX_DETAILS_TO_PUBLISH limit had been reached.

Function Level Description – DELETE_QUEUE_REC (local)

This function deletes a record from the outbound work order queue table based on a passed-in sequence number.

Function Level Description – BUILD_WODTL_OBJECT (local)

This function fetches the activity_id, unit_cost and comments for all records from tsf_wo_detail containing the passed in item and work order id. For each record found:

Populate the wooutactivity record with the activity_id, unit_cost, and comments. Then, add the wooutactivity record to the wooutactivity table.

After all details are processed, the WOOUTACTIVITY table will be added to the wooutdtl record that was passed into the function.

Function Level Description – BUILD_PACKING_OBJECT (local)

```
Procedure: BUILD_PACKING_OBJECT
    (O_error_msg              IN OUT        VARCHAR2,
     O_packing_message        IN OUT        nocopy RIB_WOOUTPACKING_TBL,
     IO_rib_wooutpacking_rec  IN OUT        nocopy RIB_WOOUTPACKING_REC,
     I_tsf_packing_id         IN            tsf_packing.tsf_packing_id%TYPE))
```

This function first constructs the RIB_WOOUTPACKFROM_REC object by fetching tsf_packing_detail.item where the tsf_packing_id matches that which was passed into the function and the record_type is 'F' (from). Once complete, add the WOOUTPACKFROM table to the wooutpacking_rec passed to the function.

Next, the RIB_WOOUTPACKTO_REC object will be constructed. Fetch the tsf_packing_detail.item where the tsf_packing_id matches that which was passed into the function and the record_type is 'R' (result). Once complete, add the WOOUTPACKTO table to the wooutpacking_rec passed to the function.

Function Level Description – LOCK_THE_BLOCK (local)

The function locks all records on the queue table for the business object. It has an O_queue_locked output that specifies whether some process other than the current process has the queue locked.

Function Level Description – HANDLE_ERRORS (local)

This procedure will handle error status values of 'H'ospital. If the LP_error_status value is 'H'ospital, it will populate the business object table with the current work order number, then create a routing info object and populate it with the sequence number of the queue record. Finally a WOOutRef object is created and added to the O_message object.

The woout_mfqueue is updated by setting the pub_status equal to API_CODES.HOSPITAL.

Trigger Impact

Create a trigger on the WO_DETAIL and TSF_HEAD to capture Inserts, Updates, and Deletes.

Trigger File Name: ec_table_thd_aiudr.trg

Table: TSFHEAD

Inserts

- Send the tsf_wo_id level info to the RMSMFM_WOOUT.ADDTOQ procedure in the MFM with the message type RMSMFM_WOOUT.HDR_ADD.

Updates

- Send the tsf_wo_id level info to the RMSMFM_WOOUT.ADDTOQ procedure in the MFM with the message type RMSMFM_WOOUT.HDR_UNAPRV.
- When a transfer is placed in 'A'pproved status the message type for this action will be outbdwocre. When a transfer's status is updated to 'D'eleted, the family manager inserts a record into the queue with a message_type = outbdwodel. When the new status is set to 'I'nput from Approved, the family manager inserts a record into the queue with message type = outbdwounaprv

Deletes

- Send the level info to the RMSMFM_WOOUT.ADDTOQ procedure in the MFM with the message type RMSMFM_WOOUT.HDR_DEL.

DTD

Here are the filenames that correspond with each message type. Please consult RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
OutBdWoCre	Work Order Create Message	WODesc.dtd
OutBdWoDel	Work Order Delete Message	WORef.dtd

Table Impact

TABLE	SELECT	INSERT	UPDATE	DELETE
WOOUT_MFQUEUE	Yes	Yes	Yes	Yes
WOPUT_PUB_INFO	Yes	Yes	Yes	Yes
TSFHEAD	Yes	No	No	No
TSF_WO_HEAD	Yes	No	No	No
TSF_WO_DETAIL	Yes	No	Yes	No
TSF_XFORM	Yes	No	No	No
TSF_XFORM_DETAIL	Yes	No	Yes	No
TSF_PACKING	Yes	No	Yes	No
TSFDETAIL	Yes	No	No	No
TSF_PACKING_DETAIL	Yes	No	No	No

Design Assumptions

- The order upon which transfer and work order messages arrive at locations participating in a multi-legged transfer does not need to be programmatically controlled.
- Work order information is never published solely at a detail level. That is, insertions, deletions and updates to work order records may not happen once the work order has been approved. In order to modify work order information, the user will need to unapprove the associated transfer. This will cause a work order header delete message to be published.
- When a work order is unapproved or deleted, header level reference information only can be published. Reference information at the detail level is not required to be published, because work order publication is never done at the individual detail level.

2

Subscription Designs

Allocation

Functional Area

Allocation

Design Overview

The allocation subscription API allows an external application to create, update, and delete allocations within RMS. The main reason for doing so is to successfully interface and track all dependent bills of lading (BOL) and receipt messages into RMS, as well as to calculate stock on hand correctly.

The allocations can be used for both stock allocations (allocating merchandise in the warehouse) and purchase order (PO), or cross-dock, allocations. The PO/cross-dock allocations can be maintained either in the database, or through the RMS application interface.

Allocation details can be created, edited, or deleted within the allocation message. Detail line items must exist on an allocation header create message for an allocation to be created. Allocation detail creates and updates will send a snapshot of the allocation record.

New item location relationships will be created for allocation detail line items entering RMS that do not previously exist within RMS.

New locations can be added to existing allocations, or current locations can be modified on existing allocations. If modifying an existing location, the passed in quantity is an adjustment to the current quantity as opposed to an overwrite.

Details can be individually removed from an allocation if the detail is not in-transit or received. An entire allocation can be deleted if none of details are in-transit or received.

The Context field is not part of the subscription messages for allocations. This field is included in the transfer/allocation publication APIs (used to define the business reason for the allocation).

Consume Module

Filename: rmssub_xallocs/b.pls

```
RMSSUB_XALLOC.CONSUME
               (O_status_code         IN OUT         VARCHAR2,
                O_error_message       IN OUT         RTK_ERRORS.RTK_TEXT%TYPE,
                I_message             IN             RIB_OBJECT,
                I_message_type        IN             VARCHAR2)
```

This procedure needs to initially ensure that the passed in message type is a valid type for Allocation messages. If the message type is invalid, a status of “E” should be returned to the external system along with an appropriate error message informing the external system that the status is invalid.

If the message type is valid, the generic RIB_OBJECT needs to be downcast to the actual object using the Oracle's treat function. If the downcast fails, a status of "E" should be returned to the external system along with an appropriate error message informing the external system that the object passed in is invalid.

If the downcast is successful, then consume needs to verify that the message passes all of the RMS business validation. It calls the RMSSUB_XALLOC_VALIDATE.CHECK_MESSAGE function to determine whether the message is valid. If the message passed RMS business validation, the function returns true, otherwise it returns false. If the message has failed RMS business validation, a status of "E" should be returned to the external system along with the error message returned from the CHECK_MESSAGE function.

Once the message has passed RMS business validation, it can be persisted to the RMS database. It calls the RMSSUB_XALLOC_SQL.PERSIST_MESSAGE() function. If the database persistence fails, the function returns false. A status of "E" should be returned to the external system along with the error message returned from the PERSIST_MESSAGE() function.

Once the message has been successfully persisted, there is nothing more for the consume procedure to do. A success status, "S", should be returned to the external system indicating that the message has been successfully received and persisted to the RMS database.

RMSSUB_XALLOC.HANDLE_ERROR() is the standard error handling function that wraps the API_LIBRARY.HANDLE_ERROR function.

Business Validation Module

It should be noted that some of the business validation is referential or involves uniqueness. This validation is handled automatically by the referential integrity constraints and the unique indexes implemented on the database.

Filename: rmssub_xallocvals/b.pls

```
RMSSUB_XALLOC_VALIDATE.CHECK_MESSAGE
               (O_error_message      IN OUT         VARCHAR2,
                O_alloc_rec          OUT            ALLOC_REC,
                I_message            IN             RIB_XAllocDesc,
                I_message_type       IN             VARCHAR2)
```

This function performs all business validation associated with message and builds the allocation record for persistence.

ALLOCATION CREATE

- Check required fields
- If item is a pack, verify receive as type is Pack for from location (warehouse).
- Verify details exist.
- Default fields (status at header, qty pre-scaled, non scale ind).
- Build allocation records.
- Perform the following steps if allocation is **not** cross-docked from an order.
  - Retrieve and build all to-locations that the item does not currently exist at.
  - Build price history records.

ALLOCATION MODIFY

- Check required fields.
- Populate record.

ALLOCATION DELETE

- Check required fields.
- Verify the allocation is not in-transit or received.
  An allocation in-transit or received may have a value (other than zero) for any of the following fields: distro quantity, selected quantity, canceled quantity, received quantity, or PO received quantity.

ALLOCATION DETAIL CREATE

- Check required fields.
- Verify details exist.
- Build allocation records.
- Perform following steps if allocation is NOT cross-docked from an order.
  - Retrieve and build all to-locations that the item does not currently exist at.
  - Build price history records.

ALLOCATION DETAIL MODIFY

- Check required fields.
- If existing allocation records are being modified:
  - Verify the allocation is not in-transit or received*.
  - Verify modification to quantity does not fall to zero or below.

ALLOCATION DETAIL DELETE

- Check required fields.
- Verify the allocation is not in-transit or received*.
- Check if deleting detail(s) removes all records from allocation. If so, process message as allocation delete.

Bulk or Single DML Module

Filename: rmssub_xallocsqls/b.pls

```
RMSSUB_XALLOC_SQL.PERSIST
              (O_error_message      IN OUT        VARCHAR2,
               I_dml_rec            IN            ALLOC_RECTYPE ,
               I_message            IN            RIB_XAllocDesc)
```

ALLOCATION CREATE

- Insert a record into the allocation header table.
- Insert a record into the allocation detail table.
- Insert a record into the allocation charge table.
- Update transfer reserved for from-location.
- Update transfer expected for to-location.
- If item is a pack, update pack component reserved qty for the from location.
- If necessary, insert a record into the item supplier country location table.
- If necessary, insert a record into the item location and stock on hand tables.
- If necessary, insert a record into the price history table.

ALLOCATION MODIFY

- Update header record (alloc desc and release date).

ALLOCATION DETAIL CREATE

- Insert a record into the allocation detail table.
- Insert a record into the allocation charge table.
- Update transfer reserved for from-location.
- Update transfer expected for to-location.
- If item is a pack, update pack component reserved qty for the from location.
- If necessary, insert a record into the item supplier country location table.
- If necessary, insert a record into the item location and stock on hand tables.
- If necessary, insert a record into the price history table.

ALLOCATION DETAIL MODIFY

- Update the allocation detail table by adjusting the existing allocated quantity using the passed in quantity. This can either increase or decrease the existing quantity.
- Update transfer reserved for from-location.
- Update transfer expected for to-location.
- If item is a pack, update pack component reserved qty for the from location.

ALLOCATION DETAIL DELETE

- Delete the record from the allocation detail table.
- Delete the record from the allocation charge table.
- Update transfer reserved for from-location.
- Update transfer expected for to-location.
- If item is a pack, update pack component reserved qty for the from location.

ALLOCATION DELETE

- Delete the record from the allocation header table.
- Delete all associated record from the allocation detail table.
- Delete all associated record from the allocation charge table.
- Update transfer reserved for from-location.
- Update transfer expected for to-location.
- If item is a pack, update pack component reserved qty for the from location.

Message DTD

Here are the filenames that correspond with each message type. Please consult RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Type	Message Type Description	DTD Name
XAllocCre	External Allocation Create	XAllocDesc.dtd
XAllocDel	External Allocation Delete	XAllocRef.dtd
XAllocDtlCre	External Allocation Detail Create	XAllocDesc.dtd
XAllocDtlDel	External Allocation Detail Delete	XAllocRef.dtd
XAllocDtlMod	External Allocation Detail Modification	XAllocDesc.dtd
XAllocMod	External Allocation Modification	XAllocDesc.dtd

Design Assumptions

- This API only applies to store level zone pricing.
- Locations are only created if an item-zone-price record already exists for the location (store as to-locations are assumed to always be stores in this API).
- This API does not currently handle inner packs when needing to create pack component location information.
- Required fields are shown in RIB documentation.
- Passed in item is at transaction level.
- From location is a stockholding location.
- Triggers impacting any allocation or item/location tables should be turned off unless deemed necessary.
- This interface does not provide functionality to manage or track the movement of inventory. Because the allocation quantities are not generated based upon RMS inventory positions, RMS provides no stock on hand or inventory validation.

RMS recommends the scheme below for allocations sequence numbers (related to ordering and transfers) in order to accommodate interfacing applications that do not distinguish between stock movement.

Orders: 1 - 99,999,999

Allocations: 1,000,000,000 - 2,999,999,999

Transfers: 3,000,000,000 - 5,999,999,999

Clients must ensure that they keep in sync with or provide their own sequence number scheme.

Tables

TABLE	SELECT	INSERT	UPDATE	DELETE
ALLOC_HEADER	Yes	Yes	Yes	Yes
ALLOC_DETAIL	Yes	Yes	Yes	Yes
ALLOC_CHRG	Yes	Yes	No	Yes
ITEM_SUPP_COUNTRY	Yes	No	No	No
ITEM_SUPP_COUNTRY_LOC	Yes	Yes	No	No
ITEM_LOC_SOH	Yes	Yes	Yes	No
STORE	Yes	No	No	No
WH	Yes	No	No	No
ITEM_LOC	Yes	Yes	No	No
SYSTEM_OPTIONS	Yes	No	No	No
ORDHEAD	Yes	No	No	No
PRICE_HIST	No	Yes	No	No
ITEM_MASTER	Yes	No	No	No

Appointments

Functional Area

Appointments subscription

Design Overview

An appointment is information about the arrival of merchandise at a location. RMS subscribes to appointment messages from the RIB that are published by an external application such as a warehouse management system (for example, RWMS). RMS processes these messages and attempts to receive against and close out the appointment. In addition, RMS attempts to close the document that is related to the appointment. A document can be a purchase order, a transfer, or an allocation.

Appointment status

Appointment messages cause the creation, updating, and closure of an appointment in RMS. Typically the processing of a message results in updating the status of an appointment in the APPT_HEAD table's status column. Valid values for the status column include:

- SC – Scheduled
- MS – Modified Scheduled
- AR – Arrived
- CL – Closed

A description of appointment processing follows.

Appointment processing

The general appointment message processes occur in this order:

1. An appointment is created for a location with a store or warehouse type from a scheduled appointment message. It indicates that merchandise is about to arrive at the location. Such a message results in a 'SC' status. At the same time, the APPT_DETAIL table is populated to reflect the purchase order, transfer, or allocation that the appointment corresponds to, along with the quantity of the item scheduled to be sent.
2. Messages that modify the earlier created appointment update the status to 'MS'.
3. Once the merchandise has arrived at the location, the appointment is updated to an 'AR' (arrived) status.
4. Another modification message that contains a receipt identifier prompts RMS to insert received quantities into the APPT_DETAIL table.
5. After all items are received, RMS attempts to close the appointment by updating it to a 'CL' status.
6. RMS will close the corresponding purchase order, transfer, or allocation 'document' if all appointments are closed.

Appointments records indicate the quantities of particular items sent to various locations within the system. The basic functional entity is the appointment record. It consists of a header and one or more detail records. The header is at the location level; the detail record is at the item-location level (with ASN as well, if applicable). Documents are stored at the detail level; a unique appointments ID is stored at the header level. In addition, a receipt number is stored at the detail level and is inserted during the receiving process within RMS.

Subscription Packages

Filename: rmssub_receivings/b.pls

```
RMSSUB_RECEIVING.CONSUME
               (O_status_code         IN OUT         VARCHAR2,
                O_error_message       IN OUT         VARCHAR2,
                I_message             IN             RIB_OBJECT,
                I_message_type        IN             VARCHAR2,
                O_rib_otbdesc_rec     OUT            RIB_OTBDESC_REC,
                O_rib_error_tbl       OUT            RIB_ERROR_TBL)
```

This procedure will make calls to receiving or appointment functions based on the value of I_message_type. If I_message type is RECEIPT_ADD or RECEIPT_UPD, then a call is made to RMSSUB_RECEIPT.CONSUME, casting the message as a RIB_RECEIPTDESC_REC. If I_message_type is APPOINT_HDR_ADD, APPOINT_HDR_UPD, APPOINT_HDR_DEL, APPOINT_DTL_ADD, APPOINT_DTL_UPD, or APPOINT_DTL_DEL, then a call is made to RMSSUB_APPOINT.CONSUME. This is the procedure called by the RIB.

```
RMSSUB_RECEIVING.HANDLE_ERRORS
               (O_status_code         IN OUT         VARCHAR2,
                IO_error_message      IN OUT         VARCHAR2,
                I_cause               IN             VARCHAR2,
                I_program             IN             VARCHAR2)
```

Standard error handling function that wraps the API_LIBRARY.HANDLE_ERROR function.

Filename: rmssub_appoints/b.pls

```
RMSSUB_APPOINT.CONSUME.CONSUME
               (O_status_code         IN OUT         VARCHAR2,
                O_error_message       IN OUT         VARCHAR2,
                I_message             IN             RIB_OBJECT,
                I_message_type        IN             VARCHAR2)
```

This function validates that the message type is valid for appointment subscription. If not, it returns a status of ‘E’ along with an error message to the calling function. If it is valid, it casts the message as RIB_APPOINTDESC_REC or RIB_APPOINTREF_REC depending on the message type, and calls local procedures HDR_ADD_CONSUME, HDR_UPD_CONSUME, HDR_DEL_CONSUME, DTL_ADD_CONSUME, DTL_UPD_CONSUME and DTL_DEL_CONSUME to perform the actual subscription logic.

APPOINTMENT CREATE

- Location must be a valid store or warehouse.
- Document must be valid based on document type (‘P’ for purchase order, ‘T’ for transfer, ‘A’ for allocations).
- Item must be a valid item.
- Insert header to APPT_HEAD if a record does not exist.
- Insert details to APPT_DETAIL if records do not already exist.

APPOINTMENT MODIFY

- Location must be a valid store or warehouse.
- Item must be a valid item.
- Update or insert into APPT_HEAD. Call APPT_DOC_CLOSE_SQL.CLOSE_DOC to close the document if the new appointment status is ‘AC’.

APPOINTMENT DELETE

- Location must be a valid store or warehouse.
- Delete both header and detail records in APPT_HEAD and APPT_DETAIL.

APPOINTMENT DETAIL CREATE

- Location must be a valid store or warehouse.
- Document must be valid based on document type (‘P’ for purchase order, ‘T’ for transfer, ‘A’ for allocations).
- Item must be a valid item.
- Insert details to APPT_DETAIL if records do not already exist.

APPOINTMENT DETAIL MODIFY

- Location must be a valid store or warehouse.
- Update or insert into APPT_DETAIL.

APPOINTMENT DETAIL DELETE

- Location must be a valid store or warehouse.
- Delete from APPT_DETAIL.

Message DTD

Here are the filenames that correspond with each message type. Please see RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
Appointcre	Appointment Create Message	AppointDesc.dtd
Appointhdrmod	Appointment Header Modify Message	AppointDesc.dtd
Appointdel	Appointment Delete Message	AppointRef.dtd
Appointdtlcre	Appointment Detail Create Message	AppointDesc.dtd
Appointdtlmod	Appointment Detail Modify Message	AppointDesc.dtd
Appointdtldel	Appointment Detail Delete Message	AppointRef.dtd

Design Assumptions

- One of the primary assumptions in the current API approach is that ease of code will outweigh performance considerations. It is hoped that the 'trickle' nature of the flow of data will decrease the need to dwell on performance issues and instead allow developers to code in the easiest and most straightforward manner.
- The adaptor is only set up to call stored procedures, not stored functions. Any public program, then, needs to be a procedure.
- Detail records may contain the same PO/item combination, differentiated only by the ASN number; however, the ASN field will be NULL for detail records which are not associated with an ASN.

Tables

TABLE	SELECT	INSERT	UPDATE	DELETE
APPT_HEAD	Yes	Yes	Yes	Yes
APPT_DETAIL	Yes	Yes	Yes	Yes
ORDHEAD	Yes	No	Yes	No
TSFHEAD	Yes	No	Yes	No
ALLOC_HEADER	Yes	No	Yes	No
STORE	Yes	No	No	No
WH	Yes	No	No	No
ITEM_MASTER	Yes	No	No	No
ORDLOC	Yes	No	No	No
DEAL_CALC_QUEUE	Yes	No	No	Yes
OBLIGATION	Yes	No	No	No
OBLIGATION_COMP	Yes	No	No	No
ALC_HEAD	Yes	No	No	Yes
ALC_COMP_LOC	Yes	No	No	Yes
V_PACKSKU_QTY	Yes	No	No	No
TSFDETAIL	Yes	No	No	No
SHIPMENT	Yes	No	No	No
SHIPSKU	Yes	No	No	No
ITEM_LOC	Yes	No	No	No
ITEM_LOC_SOH	Yes	No	Yes	No
ALLOC_DETAIL	Yes	No	No	No

ASNIN

Functional Area

Advance shipping notice (ASN) from a supplier

Business Overview

A transfer or allocation shipment is often a group of stock orders together on one truck. These multiple transfers or allocations will be grouped together using an ASN number. This number will be stored on the header record for the shipment. All shipments will be associated with an order or an ASN number now rather than an order or transfer as it worked previously.

A supplier or consolidator will send an advanced shipping notice (ASN) to software such as RWMS that publishes the information to the Oracle Retail Integration Bus (RIB). RMS subscribes to the ASN information and places the information onto RMS tables depending upon the validity of the records enclosed within the ASN message.

The ASN message will consist of a header record, a series of order records, carton records and item records. For each message, header, order and item record(s) will be required. The carton portion of the record is optional. If a carton record is present, however, then that carton record must contain items in it.

The header record will contain information about the shipment as a whole. The order records will identify which orders are associated with the merchandise being shipped. If the shipment is packed in cartons, carton records will identify which items are in which cartons. The item records will contain the items on the shipments, along with the quantity shipped. The items on the shipment should be on the ORDLOC table for the order and location specified in the header and order records.

Package Impact

Filename: rmssub_asnins/b.pls

The master Oracle Object used is called RIB_OBJECT. Messages passed into subscription packages are of the type RIB_OBJECT. All objects created in the subscription packages subclass the RIB_OBJECT type. Also, for RIB Object APIs, there is only one PL/SQL package per message family.

Public API Procedure

```
RMSSUB_ASNIN.CONSUME
               (O_STATUS_CODE         IN OUT         VARCHAR2,
                O_ERROR_MESSAGE       IN OUT         VARCHAR2,
                I_MESSAGE             IN             RIB_OBJECT,
                I_MESSAGE_TYPE        IN             VARCHAR2);
```

The following is a description of the RMSSUB_ASNIN.COMSUME procedure:

1. The public procedure checks the message type if it is create (**ASNINCRE**), modify (**ASNINMOD**), or delete (**ASNINDEL**).
2. If the message type is equivalent to **ASNINDEL** then,
   - It will get the message in the record **RIB_ASNINREF_REC**.
   - If message exists in the record then it will call the private function **PROCESS_DELETE**. It will delete the ASN record from the appropriate shipment and invoice database tables depending upon the success of the validation.
   - If no messages exist in the record then it will raise a program error that no message was deleted.
3. If the message type is equivalent to **ASNINCRE** or **ASNINMOD** then,
   - It will get the message in the record **RIB_ASNINDESC_REC**.
   - It will parse the message or records by passing on to the private function **PARSE_ASN**.
   - After parsing the records, it will check if the message type is null or not equal to ASNINCRE/ASNINMOD and if the message contains a PO record. A program error will raise if there is no or wrong message type given and if there is no PO record.
   - If the records are valid after parsing, the detail records are retrieved in a list which will loop through to several private functions and process the records.

   Inside the loop:

   a. Records are passed on to the private function **PARSE_ORDER**.
   b. Delete container and item records from the previous order.
   c. Check if CARTON_IND is equal or not equal to **`C`.**
   d. If **CARTON_IND equal to `C`** then the records will be passed on to **PARSE_CARTON** and **PARSE_ITEM** private functions. The records will then be processed by passing on to **PROCESS_ASN** private function. The records are placed in the appropriate shipment and ordering database tables depending upon the success of the validation.
   e. If **CARTON_IND is NOT equal to `C`** then a separate call to parse item (**PARSE_ITEM** private function) is required in order to retrieve those items that are not part of a container, and the records will be processed by passing on to **PROCESS_ASN** private function. The records are place in the appropriate shipment and ordering database tables depending upon the success of the validation.
   f. If there are no more records to process, the loop terminates.

Error Handling

If an error occurs in this procedure or any of the internal functions, this procedure places a call to **HANDLE_ERRORS** in order to parse a complete error message and pass back a status to the RIB.

```
HANDLE_ERRORS
          (O_status                IN OUT        VARCHAR2,
           IO_error_message        IN OUT        VARCHAR2,
           I_cause                 IN            VARCHAR2,
           I_program               IN            VARCHAR2)
```

This function is used to put error handling in one place in order to make future error handling enhancements easier to implement. All error handling in the internal **RMSSUB_ASNIN** package and all errors that occur during subscription in the ASN_SQL package (and whatever packages it calls) will flow through this function.

The function should consist of a call to API_LIBRARY.HANDLE_ERRORS. API_LIBRARY.HANDLE_ERRORS accepts a program name, the cause of the error and potentially an unparsed error message if one has been created through a call to SQL_LIB.CREATE_MESSAGE. The function uses these input variables to parse a complete error message and pass back a status, depending upon the message and error type, back up through the consume function and up to the RIB.

Private Internal Functions and Procedures

PARSE_ASN

This function will used to extract the header level information from the RIB_ASNInDesc_REC and place that information onto an internal ASN header record.

```
TYPE asn_record IS RECORD( asn                          SHIPMENT.ASN%TYPE,
                           destination
    SHIPMENT.TO_LOC%TYPE,
                           ship_date
    SHIPMENT.SHIP_DATE%TYPE,
                           est_arr_date
    SHIPMENT.EST_ARR_DATE%TYPE,
                           carrier                      SHIPMENT.COURIER%TYPE,
                           ship_pay_method      ORDHEAD.SHIP_PAY_METHOD%TYPE,
                           inbound_bol          SHIPMENT.EXT_REF_NO_IN%TYPE,
                           supplier                     ORDHEAD.SUPPLIER%TYPE,
                           carton_ind                   VARCHAR2(1));
```

PARSE_ORDER

This function will used to extract the order level information from the RIB_ASNInPO_REC/ASN number from shipment table and place that information onto an internal order record.

PARSE_CARTON

This function will used to extract the carton level information (an indicator in the ASN header record will indicate whether this information exists) from the RIB_ASNInCtn_REC / ASN and ORDER number from shipment table and place that information onto an internal arrayed carton record.

PARSE_ITEM

This function will used to extract the item level information from the RIB_ASNInItem_REC/ASN and ORDER number in the shipment table/CARTON number from carton table and place that information onto an internal arrayed item record.

Validation

PROCESS_ASN

After the values are parsed for a particular order in an ASN record, RMSSUB_ASNIN.CONSUME will call this function, which will in turn call various functions inside ASN_SQL in order to validate the values and place them on the appropriate shipment and ordering database tables depending upon the success of the validation.

Only one ASN and order record will be passed in at a time, whereas multiple cartons and items will be passed in as arrays into this function. If one order, carton or item value is rejected, then current functionality dictates that the entire ASN message will be rejected.

PROCESS_DELETE

In the event of a delete message, this function will be called rather than PROCESS_ASN. This function will take the asn_no from the parsing function and pass it into ASN_SQL in order to delete the ASN record from the appropriate shipment and invoice tables.

Message DTD

Here are the filenames that correspond with each message type. Please consult the RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
asnincre	ASN Inbound Create Message	ASNInDesc.dtd
asnindel	ASN Inbound Delete Message	ASNInRef.dtd
asninmod	ASN Inbound Modify Message	ASNInDesc.dtd

Design Assumptions

- One of the primary assumptions in the current API approach is that ease of code will outweigh performance considerations. It is hoped that the ‘trickle’ nature of the flow of data will decrease the need to dwell on performance issues and instead allow developers to code in the easiest and most straightforward manner.
- The adaptor is only set up to call stored procedures, not stored functions. Any public program then needs to be a procedure.

Tables

TABLE	SELECT	INSERT	UPDATE	DELETE
SHIPMENT	Yes	Yes	Yes	Yes
SHIPSKU	Yes	Yes	No	Yes
CARTON	No	Yes	No	Yes
INVC_XREF	No	No	No	Yes
STORE	Yes	No	No	No
WH	Yes	No	No	No
ORDHEAD	Yes	No	No	No

ASNOUT

Functional Area

ASNOUT

Design Overview

An internal advance shipment notification (ASN) message holds data that is used by RMS to create or modify a shipment record. Also known as a bill of lading (BOL), internal ASNs are published by an application that is external to RMS, such as a warehouse management system (RWMS, for example). In contrast to a BOL is the external ASN, which is generated by a supplier and shows merchandise movement from the supplier to a retailer location, like a warehouse or store. This overview describes the BOL type of advance shipment notification.

Internal ASNs are notifications to RMS that inventory is moving from one location to another. RMS subscribes to BOL messages from the Oracle Retail Integration Bus (RIB). The external application publishes these ASN messages for:

- Allocations that RMS previously initiated through a stock order message.
- Transfers that RMS previously initiated through a stock order message.
- Transfers that the external application generates itself (an RMS transfer type of 'EG' within RMS).

Individual stock orders are held on the transfer and allocation heading tables in RMS. A message may contain data about multiple transfers or allocations, and as a result, the shipment record in RMS would reflect these multiple movements of merchandise. A bill of lading number on the shipment record is a means of tracking one or more transfers and allocations back through the respective stock order and purchase order records.

BOL Message Structure

Because RMS uses a BOL message only to create a new shipment record, there is one subscribed BOL message. The message consists of a header, a series of transfers or allocations (called 'Distro' records), carton records, and item records. Thus the structure of a BOL hierarchical message would be:

- Message header – This is data about the entire shipment including all distro records, cartons, and items.
- Distro record – The individual transfer or allocation (generated earlier by RMS as a stock order number).
- Carton – Carton numbers and location. Cartons are required on all BOL messages.
- Items – Details about all items in the carton.

When external locations (stores or warehouses) ship products, they send a BOL message (otherwise known as an outbound ASN message) to let RMS know that they are shipping the stock and to let the receiving locations know that the stock is on the way. The external locations can create BOL messages for three scenarios: a transfer was requested (RMS knows about it), an allocation was requested (RMS knows about it), and on their own volition (externally generated - EG). A single BOL message can contain records generated for any or all of these transactions.

The system allows multiple transfers or allocations per shipment. This mirrors what actually happens. A stock order shipment is often a group of transfers or allocations on one truck. These multiple transfers or allocations are grouped together using a single BOL number (ASN number when coming from a supplier). This number will be stored on the header record for the shipment. All shipments will be associated with an order or a BOL number.

The BOL message is a hierarchical message that will consist of a header record, a series of distro records (transfers or allocations in RMS) inside the header record, carton records inside the distros and item records inside the cartons. The header record will contain information about the shipment as a whole. The distro records will identify which transfers or allocations are associated with the merchandise being shipped. If the shipment is packed in cartons, carton records will identify which items are in which cartons. The item records will contain the items on the shipments, along with the quantity shipped.

In terms of the data flow, an external location (store or warehouse) will publish a BOL message to the RIB. RMS will subscribe to the BOL message from the RIB, and create or modify shipments, transfers and allocations in RMS depending on the validity of the records enclosed within the message. Because the ownership of the goods moves to the receiving location at the time of shipment, stock buckets are updated and financial transaction records are written when RMS processes the BOL message.

Subscription Packages

Filename: rmssub_asnouts/b.pls

```
RMSSUB_ASNOUT.CONSUME
               (O_status_code        IN OUT        VARCHAR2,
                O_error_message      IN OUT        VARCHAR2,
                I_message            IN            RIB_OBJECT,
                I_message_type       IN            VARCHAR2)
```

This procedure will need to initially ensure that the passed in message type is a valid type for ASNOUT messages.

If the message type is invalid, a status of “E” should be returned to the external system along with an appropriate error message informing the external system that the status is invalid.

If the message type is valid, the generic RIB_OBJECT needs to be downcast to the actual object using the Oracle’s treat function. If the downcast fails, a status of “E” is returned to the external system along with an appropriate error message informing the external system that the object passed in is invalid.

If the downcast is successful, then consume will parse the message, verify that the message passes all of the RMS business validation, and persist the information to the RMS database. It does the following:

- Calls PARSE_BOL to parse the shipment level information on the message. Insert or update shipment based on the bill of lading number (bol_nbr).
- One shipment can contain multiple distros (transfers and allocations in RMS). Within each distro, call PARSE_DISTRO and PARSE_ITEM to parse and build a collection of items that are transferred or allocated.
- For break-to-sell items, if the sellable item is on the message, call CHECK_ITEMS and GET_ORDERABLE_ITEMS to convert the sellable item(s) to the corresponding orderable item(s). The orderable items will be inserted or updated on transfer/allocation and shipment tables.
- For catch weight items, validate and aggregate weight for the same item.
- Call PROCESS_DISTRO to perform business logic associated with shipping a transfer or an allocation, including insert or update transfer/allocation header and detail, insert or update SHIPSKU, move inventory to in transit buckets on ITEM_LOC_SOH, write stock ledger.
- Bulk inserts and updates are performed to improve performance.

If an error occurs in the process, a status of “E” is returned to the external system along with the failure message. Otherwise, a success status, “S”, is returned to the external system indicating that the message has been successfully received and persisted to the RMS database.

PARSE_BOL

This function parses the RIB_ASNOutDesc_Rec and builds an API bol_record for processing. It also calls RMSSUB_ASNOUT.PROCESS_BOL to check the existence of SHIPMENT based on the bol number.

PROCESS_BOL

This function calls BOL_SQL.PUT_BOL to check the existence of SHIPMENT based on the bol number.

PARSE_DISTRO

This function parses the RIB_ASNOutDesc_Rec and builds an API distro_record for processing.

PARSE_ITEM

This function builds a collection of API item_table that contains item level information for the transfer or allocation. For a simple pack catch weight item, it also aggregates the weight for the same item.

PROCESS_DISTRO

Depending on the distro type (transfer or allocation), this function calls BOL_SQL.PUT_TSF, BOL_SQL.PUT_TSF_ITEM and BOL_SQL.PROCESS_TSF or BOL_SQL.PUT_ALLOC, BOL_SQL.PUT_ALLOC_ITEM and BOL_SQL.PROCESS_ALLOC to perform the bulk of the business logic for shipping a transfer or an allocation.

CHECK_ITEMS

This function separates the item details on the message into two groups: one contains sellable items and one contains non-sellable items. The sellable items will be converted into orderable items for shipment.

GET_ORDERABLE_ITEMS

This function builds a collection of orderable items based on the sellable items. Depending on the distro type, it calls ITEM_XFORM_SQL.TSF_ORDERABLE_ITEM_INFO (for transfers) or ITEM_XFORM_SQL.ALLOC_ORDERABLE_ITEM_INFO (for allocations) to distribute the sellable quantities among the orderable items.

HANDLE_ERRORS

This function calls API_LIBRARY.HANDLE_ERRORS to perform error handling.

Filename: bolsqls/b.pls

BOL_SQL.PUT_BOL

This function checks the existence of a shipment based on the BOL number, and creates a shipment if it does not exist.

BOL_SQL.PUT_TSF

This function checks the existence of a transfer in RMS based on the transfer number and does the following:

- If the transfer exists, it updates the transfer to shipped status.
- For a 'CO' type of transfer, a customer order must be associated with it.
- If the transfer does not exist, it creates a transfer of type 'EG' (externally generated). Since the sending location is already aware of the transfer, the new transfer should **not** be published to the RIB again.

BOL_SQL.PUT_TSF_ITEM

This function checks the existence of an item on a transfer based on the transfer number and the item number. It does the following:

- If the input item is a referential item, fetch and use its transactional level item.
- If the item exists on the transfer, update the quantity buckets on TSFDETAIL.
- If the item does **not** exist on the transfer, create TSFDETAIL. However, new items cannot be added to a closed transfer.
- If sending a pack from a warehouse, reject the message if the sending location does not stock packs, unless the sending location is a finisher.
- For an 'EG' type of transfer, physical location is on the transfer. Distribute the transferred quantity to its virtual locations by creating an inventory flow structure and save it on SHIPITEM_INV_FLOW.

BOL_SQL.PROCESS_TSF

This function calls BOL_SQL.SEND_TSF to perform the bulk of the transfer shipment business logic. It does the following:

- If the sending location of the transfer is a finisher, this is the second leg of a multi-legged transfer. Call TSF_WO_COMP_SQL.WO_ITEM_COMP to perform any necessary item transformations, including adjusting inventory and average cost of the old and new items, and writing TRAN_DATA for the adjusted inventory.
- Update inventory (stock_on_hand and tsf_reserved_qty) for the item transferred at the sending location.
- Update inventory (in_transit_qty and tsf_expected_qty) and average cost for the item transferred at the receiving location.

- When the item shipped is a pack item, if the pack item is stocked as a pack at the sending/receiving location, inventory is updated for both the pack item (stock_on_hand, tsf_reserved_qty, in_transit_qty, tsf_expected_qty) and the pack component items (pack_comp_soh, pack_comp_resv, pack_comp_intran, pack_comp_exp). On the other hand, if the pack item is **not** stocked as a pack at the sending/receiving location, inventory is updated for the pack component items only (stock_on_hand, tsf_reserved_qty, in_transit_qty, tsf_expected_qty).
- When the item shipped is a simple pack catch weight item, average weight on ITEM_LOC_SOH is updated if the pack is stocked as a pack at the sending/receiving location.
- When the item shipped is a simple pack catch weight item and the pack component's standard UOM is a mass UOM (for example, OZ), component's inventory is updated by the actual weight shipped.
- Call STKLEDGR_SQL.WRITE_FINANCIALS to write TRAN_DATA records for the sending and receiving locations:
  - 37/38 – for inter-company transfer in/out, in which case the sending and receiving locations belong to different transfer entities. The transfer is valued at the transfer price on TSFDETAIL. Transfer price is defined in the sending location's currency.
  - 17/18 – for inter-company markup/markdown. It records the total retail difference between the transfer price and the sending location's unit retail. It is written against the sending location.
  - 30/32 – for intra-company transfer in/out, in which case the sending and receiving locations belong to the same transfer entity. The transfer is valued at the transfer cost on TSFDETAIL if defined. If not, it is valued at the sending location's WAC. WAC is dependent on the accounting method used, which could be retail accounting or standard cost accounting or average cost accounting. Both WAC and transfer cost are in the sending location's currency.
  - 11/13 – for intra-company markup/markdown. It records the total retail difference between the sending and receiving locations. It is written against either the sending or the receiving location, depending on the settings on the system options (tsf_md_store_to_store_snd_rcv, tsf_md_wh_to_store_snd_rcv, tsf_md_store_to_wh_snd_rcv, tsf_md_wh_to_wh_snd_rcv).
  - 71/72 – for intra-company cost variance. It records the total cost variance as a result of the difference between the sending location's WAC and the transfer cost. It is written against the sending location.
  - 65 – for transfer restocking fees if a restocking percentage is defined on the transfer detail. It can be for an inter-company or an intra-company transfer. It is written against the sending locations.
  - 28 – for up charges.
  - When a deposit content item is shipped, a TRAN_DATA record is also written for the container item for tran codes 30/32 and 37/38. The total cost should be based on the cost of the container.

Note: When a simple pack catch weight item is shipped, the total cost is evaluated at the weight shipped. As a result, TRAN_DATA.total_cost reflects the weight shipped for tran codes 37/38, 30/32, 71/72 and 65. However, all the retail calculation is not weight-based. As a result, TRAN_DATA.total_retail and tran codes 17/18, 11/13 do **not** reflect the actual weight.

- Creates shipsku for the item. For a simple pack catch weight item, weight_expected and weight_expected_uom are written along with the qty_expected.
- shipsku.unit_retail is the sending location's unit retail. When a break to sell orderable item is shipped, its unit retail is derived from its sellable items. Similarly, in a multi-legged transfer scenario, the sending location can be a finisher. Because a finisher does not have unit retail, the unit retail at the receiving location is used.
- For a customer order transfer that is shipped directly to the customer, call STOCK_ORDER_RCV_SQL.TSF_LINE_ITEM to receive the shipment.

BOL_SQL.PUT_ALLOC

This function checks the existence of an allocation based on the allocation number, item number and warehouse. If the input item is a referential item, its transactional level item is used. Reject the message if the allocation does not exist.

BOL_SQL.PUT_ALLOC_ITEM

This function checks the existence of allocation detail based on the allocation number and the receiving location. Because goods can only be allocated to stores, reject the message if the receiving location is a warehouse. It does the following:

- If the store exists on allocation detail, update the quantity buckets on ALLOC_DETAIL.
- If the store does **not** exist on allocation detail, create ALLOC_DETAIL.

BOL_SQL.PROCESS_ALLOC

This function calls BOL_SQL.SEND_ALLOC to perform the bulk of the allocation shipment business logic. It does the following:

- Update inventory (stock_on_hand and tsf_reserved_qty) for the item allocated at the sending location.
- Update inventory (in_transit_qty and tsf_expected_qty) and average cost for the item allocated at the receiving location.
- When the item shipped is a pack item, if the pack item is stocked as a pack at the sending/receiving location, inventory is updated for both the pack item (stock_on_hand, tsf_reserved_qty, in_transit_qty, tsf_expected_qty) and the pack component items (pack_comp_soh, pack_comp_resv, pack_comp_intran, pack_comp_exp). On the other hand, if the pack item is **not** stocked as a pack at the sending/receiving location, inventory is updated for the pack component items only (stock_on_hand, tsf_reserved_qty, in_transit_qty, tsf_expected_qty).
- When the item shipped is a simple pack catch weight item, average weight on ITEM_LOC_SOH is updated if the pack is stocked as a pack at the sending/receiving location.
- When the item shipped is a simple pack catch weight item and the pack component's standard UOM is a mass UOM (for example, OZ), component's inventory is updated by the actual weight shipped.

- Call STKLEDGR_SQL.WRITE_FINANCIALS to write TRAN_DATA records for the sending and receiving locations:
  - 37/38 – for inter-company allocation in/out, in which case the sending and receiving locations belong to different transfer entities. Allocations are valued at the sending location's WAC.
  - 30/32 – for intra-company allocation in/out, in which case the sending and receiving locations belong to the same transfer entity. Allocations are valued at the sending location's WAC.
  - 11/13 – for intra-company markup/markdown. It records the total retail difference between the sending and receiving locations. It is written against either the sending or the receiving location, depending on the settings on the system options (tsf_md_store_to_store_snd_rcv, tsf_md_wh_to_store_snd_rcv, tsf_md_store_to_wh_snd_rcv, tsf_md_wh_to_wh_snd_rcv).
  - 28 – for up charges.
  - When a deposit content item is shipped, a TRAN_DATA record is also written for the container item for tran codes 30/32 and 37/38. The total cost should be based on the cost of the container.

    Note: Similar to shipping a transfer, the retail values are not weight-based for a simple pack catch weight item.
- Creates shipsku for the item. For a simple pack catch weight item, weight_expected and weight_expected_uom are written along with the qty_expected.

Message DTD

Here are the filenames that correspond with each message type. Please consult RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
asnoutcre	ASN Outbound Create Message	ASNOutDesc.dtd

Design Assumptions

- The ASNOut subscription process supports the break to sell functionality. Transfers, allocations and shipments in RMS will only contain break to sell orderable items. Inventory adjustment and stock ledger will be performed on the orderable only, not the sellable.
- The ASNOut subscription process supports the catch weight functionality. It is assumed that a break to sell sellable item cannot be a simple pack catch weight item.
- Catch weight functionality is not completely rounded out in this release. For instance, it is **not** applied to the following areas:
  - Any of the retail calculations (including total_retail on TRAN_DATA and retail markup/markdown);
  - The total amount on SUP_DATA;
  - Open to buy buckets;
  - When a catch weight component item's standard UOM is a MASS UOM, TRAN_DATA.units is based on V_PACKSKU_QTY.qty instead of the actual weight.
- An externally generated transfer will contain physical locations. When system options INTERCOMPANY_TSF_IND = 'Y', the stock order receiving process currently does **not** support the receiving of an externally generated transfer that involves a warehouse to warehouse transfer. This is because a physical location does **not** have transfer entities.

Tables

TABLE	SELECT	INSERT	UPDATE	DELETE
TSFHEAD	Yes	Yes	Yes	No
TSFDETAIL	Yes	Yes	Yes	No
TRANSFERS_PUB_INFO	No	Yes	No	No
ALLOC_HEADER	Yes	Yes	Yes	No
ALLOC_DETAIL	Yes	Yes	Yes	No
SHIPMENT	Yes	Yes	Yes	No
SHIPSKU	Yes	Yes	Yes	No
TRAN_DATA	No	Yes	No	No
ITEM_LOC_HIST	No	Yes	No	No
ITEM_LOC_SOH	Yes	Yes	Yes	No
ITEM_LOC	Yes	Yes	No	No
ITEM_ZONE_PRICE	Yes	Yes	No	No
PRICE_HIST	No	Yes	No	No
SHIPITEM_INV_FLOW	No	Yes	No	No

TABLE	SELECT	INSERT	UPDATE	DELETE
STORE	Yes	No	No	No
WH	Yes	No	No	No
ITEM_MASTER	Yes	No	No	No
V_PACKSKU_QTY	Yes	No	No	No
ITEM_XFORM_HEAD	Yes	No	No	No
ITEM_XFORM_DETAIL	Yes	No	No	No
TSF_XFORM	Yes	No	No	No
TSF_XFORM_DETAIL	Yes	No	Yes	No
TSF_ITEM_COST	Yes	No	Yes	No
TSF_ITEM_WO_COST	Yes	No	No	No
WO_ACTIVITY	Yes	No	No	No
INV_ADJ_REASON	Yes	No	No	No
INV_ADJ	Yes	No	No	No
INV_STATUS_QTY	Yes	Yes	Yes	Yes
DEPS	Yes	No	No	No
CURRENCIES	Yes	No	No	No
CURRENCY_RATES	Yes	No	No	No
PERIOD	Yes	No	No	No
SYSTEM_OPTIONS	Yes	No	No	No
WEEK_DATA	Yes	No	No	No
MONTH_DATA	Yes	No	No	No
ITEM_SUPP_COUNTRY	Yes	No	No	No
ITEM_SUPP_COUNTRY_DIM	Yes	No	No	No
UOM_CLASS	Yes	No	No	No

Clearance

Functional Area

Clearance

Design Overview

When RMS is not the system of record for managing clearance price changes, it can take advantage of a clearance subscription API. The clearance subscription keeps RMS in sync with the external system that is responsible for maintaining clearance prices. Clearance prices are updated for item/locations that already exist in RMS. The subscription does not create or delete item/location records in RMS tables.

Clearance price changes can be performed at the following levels of the organization hierarchy: chain, area, region, district, and store. Clearance prices are updated for all stores within the location group unless marked as exceptions. Because warehouses are not part of the organization hierarchy, they are only impacted by clearance price changes applied at the warehouse level.

The subscription does not create clearance price change events; it updates the price of an item in real time.

The following rules are used to determine which stores are eligible for the clearance:

1. All stores in the location group based on the organization hierarchy (chain, area, region district);
2. Of the stores within the location group, those that have the same local currency as specified on the clearance message;
3. Of the stores with the same local currency, those in the same country as specified on the clearance message;
4. Of the remaining stores, those not found in the exception list.

The approach taken for the clearance subscription API is similar to that of the price change subscription API. The notable differences are as follows:

- On the ITEM_LOC table, the clear_ind field is updated to Y (Yes), which indicates the item is currently on clearance.
- Clearances cannot be applied to pack items.

RMS exposes an API that will allow external systems to update unit price within RMS.

This RMS API subscribes to external clearance modify messages for the purpose of integrating external clearances maintained in an external system into RMS. It updates unit prices in RMS and writes price history.

At least one detail is required for a message to be valid.

Item, item parent, item parent/differentiator, organizational hierarchy, country, and currency can be used to maintain clearances. If clearances are created using the organizational hierarchy (area, region, and so on), then the country and currency filter the list of stores the clearance will affect. Exception stores can be used to further limit stores impacted by clearance.

If any locations (zones) do not exist on item-loc, those locations will not be processed and not fail the message. When processing using either (or both) an item parent/organization hierarchy, records not found will not be processed. However, if creating clearances by transactional level item/single location, then records not found on item-loc will error out.

Clearances can only update the single unit retail. Clearances will not update item-zone-price records. No packs can be put on clearances.

Any time the system is discussing warehouse locations, those locations must be stockholding warehouses (virtual warehouse in multi-channel environment, physical in non-multi-channel environment).

This API only supports location (store) level zone pricing, that is, a zone is equivalent to a location. In addition, this API only supports warehouse retail (RMS system_options.sor_pricing_ind = 'N').

Consume Module

Filename: rmssub_xclearances/b.pls

```
RMSSUB_XCLEARANCE.CONSUME
               (O_status_code        IN OUT        VARCHAR2,
                O_error_message      IN OUT        RTK_ERRORS.RTK_TEXT%TYPE,
                I_message            IN            RIB_OBJECT,
                I_message_type       IN            VARCHAR2)
```

This procedure needs to initially ensure that the passed in message type is a valid type for clearance messages. There is only one valid message type for clearance messages, xclearancemod. If the message type is invalid, a status of "E" should be returned to the external system along with an appropriate error message informing the external system that the status is invalid.

If the message type is valid, the generic RIB_OBJECT needs to be downcast to the actual object using the Oracle's treat function. If the downcast fails, a status of "E" should be returned to the external system along with an appropriate error message informing the external system that the object passed in is invalid.

If the downcast is successful, then consume needs to verify that the message passes all of the RMS business validation. It calls the RMSSUB_XCLEARANCE_VALIDATE.CHECK_MESSAGE function to determine whether the message is valid. If the message passed RMS business validation, then the function will return true; otherwise, it will return false. If the message has failed RMS business validation, a status of "E" should be returned to the external system along with the error message returned from the CHECK_MESSAGE function.

Once the message has passed RMS business validation, it can be persisted to the RMS database. It calls the RMSSUB_XCLEARANCE_SQL.PERSIST () function. If the database persistence fails, the function returns false. A status of "E" should be returned to the external system along with the error message returned from the PERSIST() function.

Once the message has been successfully persisted, there is nothing more for the consume procedure to do. A success status, "S", should be returned to the external system indicating that the message has been successfully received and persisted to the RMS database.

Business Validation Module

Filename: rmssub_xclearancevals/b.pls

It should be noted that some of the business validation is referential or involves uniqueness. This validation is handled automatically by the referential integrity constraints and the unique indexes implemented on the database.

```
RMSSUB_XCLEARANCE_VALIDATE.CHECK_MESSAGE
           (O_error_message   IN OUT         RTK_ERRORS.RTK_TEXT%TYPE,
            O_clearance_rec   OUT            RMSSUB_XPRICECHG.PRICE_CHANGE_REC,
            I_message         IN             RIB_XClearanceDesc)
```

This function performs all business validation associated with message and builds the clearance record for persistence.

CLEARANCE MODIFY

- Check required fields.
- Validate passed-in fields (currency, country, UOM, hierarchy level).
- Verify item is above transaction level and approved.
- If diff IDs are passed in, verify they are valid for passed in item.
- Verify item passed in is not a pack.
- Validate the UOM passed in is of the same UOM class as the standard UOM.
- Convert the new retail passed in from the UOM on the message to item's standard UOM.
- POPULATING RECORD
  - Retrieve item's transaction level children if the passed in item is a parent.
  - Retrieve all locations based on passed in hierarchy type and value, currency and country.
  - Exclude locations passed in as exception stores.
  - Build clearance records.
- Populate record with message data.

Bulk or single DML module

Filename: rmssub_xclearancesqls/b.pls

Insert, update and delete SQL statements are located in package PRICING_SQL. The private functions call these packages.

This API calls STKLEDGR_PRICING_SQL package to write stock ledger for clearance, which is optimized for performance.

```
RMSSUB_XCLEARANCE_SQL.PERSIST
           (O_error_message   IN OUT         RTK_ERRORS.RTK_TEXT%TYPE,
            O_clearance_rec   OUT            RMSSUB_XPRICECHG.PRICE_CHANGE_REC)
```

CLEARANCE MODIFY

- Update the unit retail on the ITEM_LOC table for all item/locations if the single unit retail is changed and the item is not a pack. No update for multi-unit retail. Set clearance indicator to 'Y'es.
- Insert into price history all records that have item_loc relations, including sellable packs.
- Insert TRAN_DATA for transactional level items that have ITEM_LOC records and if the old standard unit retail and the new standard unit retail are NOT the same. Do **not** insert tran_data if item/location's stock_on_hand+in transit quantity is 0.
- For each TRAN_DATA record inserted, a SUP_DATA record will also be inserted.

Message DTD

Here are the filenames that correspond with each message type. Please consult the RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Type	Message Type Description	DTD Name
Xclearancemod	External Clearance Modify	XclearanceDesc.dtd

Design Assumptions

- Required fields are shown in the RIB documentation.
- Each clearance message should be committed separately. A few global temporary tables are utilized in this API to help performance. Records on these global temporary tables will be deleted on commit. This API does **not** delete from these tables.
- This API only supports location (store) level zone pricing (zone ID is equivalent to a location).
- This API only supports warehouse retail (RMS system_options.sor_pricing_ind = 'N'). As a result, warehouse retail is held on the ITEM_LOC table in warehouse currency.
- If more than one clearance occurs for a day on an item/location, then price history is updated, as opposed to a new record's being inserted for the second clearance.
- In the current implementation of this API, POS_MODS are **not** written for clearance.
- The approach taken for the clearance API is very similar to the price change API approach. The notable differences are as follows:
  - ITEM_ZONE_PRICE records is not updated
  - Clearance indicator in ITEM_LOC is updated to 'Y'
  - Clearance cannot be applied to a pack item
  - Clearance only deals with single unit retail, whereas price change can be for single unit retail, or multi unit retail or both. That drives the "TRAN_TYPE" on PRICE_HIST table.
  - When writing a tran_data record, tran_code is always "Clearance Markdown" for clearance, whereas it could be markdown or markup or markdown cancel for price change.

- Triggers impacting item/location tables should be turned off unless deemed necessary.
- This API cannot be used in when using the RPM system for price management.

Tables

TABLE	SELECT	INSERT	UPDATE	DELETE
ITEM_LOC	Yes	No	Yes	No
ITEM_LOC_SOH	Yes	No	No	No
STORE	Yes	No	No	No
WH	Yes	No	No	No
ITEM_MASTER	Yes	No	No	No
DIFF_GROUP_HEAD	Yes	No	No	No
DIFF_GROUP_DETAIL	Yes	No	No	No
CHAIN	Yes	No	No	No
AREA	Yes	No	No	No
REGION	Yes	No	No	No
DISTRICT	Yes	No	No	No
CURRENCIES	Yes	No	No	No
COUNTRY	Yes	No	No	No
PRICE_HIST	No	Yes	No	No
SUP_DATA	No	Yes	No	No
SUP	Yes	No	No	No
TRAN_DATA	No	Yes	No	No
CLASS	Yes	No	No	No
VAT_ITEM	Yes	No	No	No
PERIOD	Yes	No	No	No
SYSTEM_OPTIONS	Yes	No	No	No
API_ITEM_TEMP	Yes	Yes	No	No
API_LOC_TEMP	Yes	Yes	No	No
API_ITEM_LOC_TEMP	Yes	Yes	No	No
API_PRICE_HIST_TEMP	Yes	Yes	No	No
API_VAT_TEMP	Yes	Yes	No	No

CO Return Sale

Functional Area

Customer Order Return Sale Subscription.

Business Overview

RMS subscribes to customer order return sale messages that originate from an integration subsystem and are published to the RIB. Message processing records the inventory and financial transactions associated with customer order return sales.

The customer order return sale message contains information about the following:

- Item returned.
- The number of units returned.
- The stockholding warehouse location to which the item was returned.
- The virtual store that will be charged with the return.

When the message is processed in RMS, the item inventory at the warehouse is incremented by the number of units returned. This processing reflects the receipt of merchandise into the warehouse when the customer returns items. The item inventory at the virtual store is decremented by the number of units returned. This processing ensues because subsequent steps in the return process will record a return against this virtual store, thereby incrementing this reduced inventory.

This movement of inventory from the virtual store to the stockholding warehouse also results in the recording of a transfer financial transaction to the stock ledger.

Package Impact

Filename: rmssub_ custretsales/b.pls

```
PROCEDURE CONSUME
               (O_status_code         IN OUT         VARCHAR2,
                O_error_message       IN OUT         RTK_ERRORS.RTK_TEXT%TYPE,
                I_message             IN             RIB_OBJECT,
                I_message_type        IN             VARCHAR2)
```

This procedure calls various functions within the corresponding VALIDATE and SQL packages.

Before calling any functions, this procedure narrows I_message down to the specific object being used, depending on the message_type. For example, a ‘Cre’ or ‘Mod’ message type usually means a ‘Desc’ object is being used. A ‘Del’ message usually means a ‘Ref’ object is being used. Object narrowing is performed using the TREAT function. If the narrowing fails, this function should return a error message to the RIB stating that the object is not valid for this message family.

CONSUME first calls the family’s VALIDATE package to validate the contents of the message. The family’s SQL package is then called to perform DML.

Business Validation Mode

Filename: rmssub_custretsalevals/b.pls

```
FUNCTION CHECK_MESSAGE
          (O_error_message   IN OUT         RTK_ERRORS.RTK_TEXT%TYPE,
           O_custsale_rec    IN OUT         RMSSUB_CUSTSALE.CUSTSALE_REC_TYPE,
           I_message         IN             RIB_CUSTSALEDESC_REC,
           I_message_type    IN             VARCHAR2)
```

This function first calls the CHECK_FIELDS function to make sure all required fields are not NULL. The function then calls other functions as needed to validate all of the information that has been passed to it from the RIB.

DML Module

Filename: rmssub_custretsalesqls/b.pls

```
FUNCTION PERSIST
          (O_error_message   IN OUT  RTK_ERRORS.RTK_TEXT%TYPE,
           I_message_type    IN      VARCHAR2,
           I_custretsale_rec IN      RMSSUB_CUSTRETSALE.CUSTRETSALE_REC_TYPE)
```

This function performs the inventory and financial transactions associated with the customer order return sale. The inventory is moved, using a book transfer, from the store location to the warehouse location provided in the message. In addition a transfer financial transaction is written to the stock ledger.

Message DTD

Here are the filenames that correspond with each message type. Please consult Oracle Retail Integration Bus documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
CustretsaleCre	Customer Order Return Create Message	CustretsaleDesc.dtd

Design Assumptions

The subscriber makes some assumptions about the publisher's ability to maintain data integrity. The subscriber does not check for duplicate Create messages. It will not check for missing messages, because it has no way of knowing what would be missing. It also assumes that messages are sent in the correct sequence.

Tables

TABLE	SELECT	INSERT	UPDATE	DELETE
ITEM_MASTER	YES	NO	NO	NO
ITEM_LOC	YES	NO	YES	NO
ITEM_LOC_SOH	YES	NO	YES	NO
ITEM_LOC_HIST	YES	YES	YES	NO
STORE	YES	NO	NO	NO
V_PACKSKU_QTY	YES	NO	NO	NO

CO Sales

Functional Area

Customer Order Sales Subscription.

Business Overview

RMS subscribes to customer order sale messages that originate from an integration subsystem and are published to the RIB. Message processing records the inventory and financial transactions associated with the customer order sales.

The customer order sale message contains information about the item sold, the number of units sold, the stockholding warehouse location from which the item was shipped to the customer, the virtual store which will be credited with the sale, and the date of the sale.

When the message is processed in RMS, the item inventory at the warehouse is decremented by the number of units sold. This processing reflects the shipment of merchandise out of the warehouse to the customer. The item inventory at the virtual store is incremented by the number of units sold. This processing ensues because subsequent steps in the sale process will record a sale against this virtual store, thereby decrementing this added inventory.

This movement of inventory from the stockholding warehouse to the virtual store also results in the recording of a transfer financial transaction to the stock ledger.

Package Impact

Filename: rmssub_custsales/b.pls

```
RMSSUB_CUSTSALE.CONSUME
                  (O_status_code     IN OUT         VARCHAR2,
                   O_error_message   IN OUT         RTK_ERRORS.RTK_TEXT%TYPE,
                   I_message         IN             RIB_OBJECT,
                   I_message_type    IN             VARCHAR2);
```

CONSUME simply calls different functions within the corresponding VALIDATE and SQL packages.

Before calling any functions, CONSUME narrows I_message down to the specific object being used, depending on the message_type. For example, a ‘Cre’ or ‘Mod’ message type usually means a ‘Desc’ object is being used. A ‘Del’ message usually means a ‘Ref’ object is being used. Object narrowing is done using the TREAT function. If the narrowing fails, the CONSUME function should return an error message to the RIB stating that the object is not valid for this message family.

Business Validation Module

Filename: rmssub_custsalevals/b.pls

```
RMSSUB_CUSTSALE_VALIDATE.CHECK_FIELDS
               (O_error_message      IN OUT         RTK_ERRORS.RTK_TEXT%TYPE,
                I_message            IN             RIB_CUSTSALEDESC_REC,
                I_message_type       IN             VARCHAR2);
```

This function first calls the CHECK_FIELDS function to make sure all required fields are not NULL. Then, the function will call other functions as needed to validate all of the information that has been passed to it from the RIB.

DML Module

Filename: rmssub_custsalesqls/b.pls

```
MSSUB_CUSTSALE_SQL. PERSIST
               (O_error_message       IN OUT  RTK_ERRORS.RTK_TEXT%TYPE,
                I_message_type        IN      VARCHAR2,
                I_custsale_rec        IN      RMSSUB_CUSTSALE.CUSTSALE_REC_TYPE);
```

This function will perform the inventory and financial transactions associated with the customer order sale. The inventory is moved from the warehouse location to the store location provided in the message. The average cost at the store is updated, and any ongoing stock count snapshots are updated. In addition, a transfer financial transaction is written to the stock ledger.

Message DTD

Here are the filenames that correspond with each message type. Please consult the RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
custsalecre	Customer Sale Create Message	CustSaleDesc.dtd

Design Assumptions

The subscriber makes some assumptions with the publisher's ability to maintain data integrity. The subscriber will not check for duplicate create messages. It will not check for missing messages, because it has no way of knowing what would be missing. It also assumes that messages are sent in the correct sequence.

Tables

TABLE	SELECT	INSERT	UPDATE	DELETE
ITEM_MASTER	YES	NO	NO	NO
ITEM_LOC	YES	NO	YES	NO
ITEM_LOC_SOH	YES	NO	YES	NO
ITEM_LOC_HIST	YES	YES	YES	NO
STORE	YES	NO	NO	NO
V_PACKSKU_QTY	YES	NO	NO	NO

COGS

Functional Area

COGS Subscription

Business Overview

The cost of goods sold (COGS) interface lets a retailer make replacements, which are similar to exchanges. However, replacements involve a different accounting process than exchanges. In a replacement, a retailer replaces a previously purchased item with an equivalent unit. To make this replacement, first the retailer places the request and ships the undesirable unit out; then the replacement unit is shipped to the retailer. In RMS, the cost of goods sold interface allows the retailer to make this replacement despite the fact that the exchange is not made simultaneously.

The interface writes the value of the transaction to the transaction data tables. An external system (such as Oracle Retail Data Warehouse) can then extract that data.

COGS Messages and TRAN_DATA

The subscription process for COGS adjustments involves an interface which contains the item, location, quantity, date, order header media, order line media, and a reason code. These records are inserted into the TRAN_DATA table to affect the stock ledger. Message processing includes a call to STKLEDGER_SQL.TRAN_DATA_INSERT to insert the new transaction to the TRAN_DATA table.

RMS subscribes to integration subsystem COGS messages. This process records the inventory and financial transactions associated with a cost of goods sold message.

Package Impact

Filename: rmssub_cogsb/s.pls

```
PROCEDURE CONSUME
               (O_status_code        IN OUT        VARCHAR2,
                O_error_message      IN OUT        RTK_ERRORS.RTK_TEXT%TYPE,
                I_message            IN            RIB_OBJECT,
                I_message_type       IN            VARCHAR2)
```

CONSUME simply calls different functions within the corresponding VALIDATE and SQL packages.

Before calling any functions, CONSUME narrows I_message down to the specific object being used, depending on the message_type. For example, a ‘Cre’ or ‘Mod’ message type usually means a ‘Desc’ object is being used. A ‘Del’ message usually means a ‘Ref’ object is being used. Object narrowing is done using the TREAT function. If the narrowing fails, then the CONSUME function should return an error message to the RIB stating that the object is not valid for this message family.

CONSUME first calls the family’s VALIDATE package to validate the contents of the message. The family’s SQL package is then called to perform DML.

Business Validation Mode

Filename: rmssub_cogsvalb/s.pls

This function first calls the CHECK_FIELDS function to make sure all required fields are not NULL. Then, the function will call other functions as needed to validate all of the information that has been passed to it from the RIB.

DML Module

Filename: rmssub_cogssqlb/s.pls

```
PERSIST
              (O_error_message       IN OUT         RTK_ERRORS.RTK_TEXT%TYPE,
               I_message_type        IN             VARCHAR2,
               I_cogs_rec            IN             RMSSUB_COGS.COGS_REC_TYPE)
```

This function will perform the inventory and financial transactions associated with the COGS transaction. The inventory is adjusted at the store location based on the reason code (replacement in/out) provided in the message. In addition a net sale and permanent markdown financial transaction is written to the stock ledger.

Message DTD

Here are the filenames that correspond with each message type. Please consult the RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
CogsCre	COGS Create Message	CogsDesc.dtd

Design Assumptions

The subscriber makes some assumptions about the publisher's ability to maintain data integrity. The subscriber does not check for duplicate Create messages. It will not check for missing messages because it has no way of knowing what would be missing. It also assumes that messages are sent in the correct sequence.

Tables

TABLE	SELECT	INSERT	UPDATE	DELETE
ITEM_LOC	Yes	No	No	No
ITEM_LOC_SOH	No	No	Yes	No
TRAN_DATA	No	Yes	No	No

Cost Change

Functional Area

Cost Change

Design Overview

Cost changes can be performed at the following levels of the organization hierarchy: chain, area, region, district, and store. Unit costs are updated for all stores within the location group. Because warehouses are not part of the organization hierarchy, they are only impacted by cost changes applied at the warehouse level.

The subscription does not create cost change events; it updates the cost of an item in real time.

The cost change subscription updates unit costs for item/locations that already exist in RMS. It does not create or delete item/locations on RMS tables.

RMS exposes an API that will allow external systems to update unit cost within RMS.

This RMS API subscribes to external cost change modify messages for the purpose of integrating external cost changes maintained in an external system into RMS. It updates unit costs in RMS and writes cost history.

Consume Module

Filename: rmssub_xcostchgs/b.pls

```
RMSSUB_XCOSTCHG.CONSUME
               (O_status_code        IN OUT        VARCHAR2,
                O_error_message      IN OUT        RTK_ERRORS.RTK_TEXT%TYPE,
                I_message            IN            RIB_OBJECT,
                I_message_type       IN            VARCHAR2)
```

This procedure needs to initially ensure that the passed-in message type is a valid type for cost change messages. There is only one valid message type for Cost change messages, XCostchgMod. If the message type is invalid, a status of “E” should be returned to the external system along with an appropriate error message informing the external system that the status is invalid.

If the message type is valid, the generic RIB_OBJECT needs to be downcast to the actual object using the Oracle’s treat function. If the downcast fails, a status of “E” should be returned to the external system along with an appropriate error message informing the external system that the object passed in is invalid.

If the downcast is successful, then consume needs to verify that the message passes all of the RMS business validation. It calls the RMSSUB_XCOSTCHG_VALIDATE.CHECK_MESSAGE function to determine whether the message is valid. If the message passed RMS business validation, then the function will return true; otherwise, it will return false. If the message has failed RMS business validation, a status of “E” should be returned to the external system along with the error message returned from the CHECK_MESSAGE function.

Once the message has passed RMS business validation, it can be persisted to the RMS database. It calls the RMSSUB_XCOSTCHG_SQL.PERSIST_MESSAGE() function. If the database persistence fails, the function returns false. A status of “E” should be returned to the external system along with the error message returned from the PERSIST_MESSAGE() function.

Once the message has been successfully persisted, there is nothing more for the consume procedure to do. A success status, “S”, should be returned to the external system indicating that the message has been successfully received and persisted to the RMS database.

RMSSUB_XCOSTCHG.HANDLE_ERROR() – This is the standard error handling function that wraps the API_LIBRARY.HANDLE_ERROR function.

Business Validation Module

It should be noted that some of the business validation is referential or involves uniqueness. This validation is handled automatically by the referential integrity constraints and the unique indexes implemented on the database.

Filename: rmssub_xcostchgvals/b.pls

```
RMSSUB_XCOSTCHG_VALIDATE.CHECK_MESSAGE
               (O_error_message       IN OUT         VARCHAR2,
                O_cost_change_rec     OUT            COST_CHANGE_REC,
                I_message             IN             RIB_XCostChgDesc,
                I_message_type        IN             VARCHAR2)
```

This function performs all business validation associated with message and builds the cost change record for persistence.

COST CHANGE MODIFY

- Check required fields.
- Verify suppliers currency.
- Verify item status.
- If diff IDs are passed in, verify they are valid for passed in item.
- Verify item passed in is not a buyer pack.
- POPULATING RECORD
  - Retrieve item’s transaction level children if the passed-in item is a parent.
  - Retrieve all locations based on passed in hierarchy type and value.
  - Determine if a location to be updated is the primary location; if so, retrieve the item-supplier-country record to be updated.
  - Retrieve all item/location combinations where passed-in supplier/country is the primary supplier/country at an item location.
  - Retrieve all orderable buyer packs that the passed-in item, or its children if above transaction level.
  - If the recalculate order indicator is ‘Y’, retrieve all item/locations on approved (and worksheet) orders.
- Populate record with message data.

Bulk or Single DML Module

Filename: rmssub_xcchgsqls/b.pls

```
RMSSUB_XCOSTCHG_SQL.PERSIST
               (O_error_message       IN OUT         VARCHAR2,
                I_dml_rec             IN             COST_CHANGE_RECTYPE ,
                I_message             IN             RIB_XCostChgDesc)
```

COST CHANGE

- Update the unit cost on item supplier country location table for all item/locations.
- If one of the locations was a primary location, update the item supplier country table. Insert into price history all records for all item/locations related to the supplier/country as the primary supplier/country.
- If average cost method is not used (system option ECL_IND = N), update the unit cost on item location stock on hand table for all item/locations related to the supplier/country as the primary supplier/country *(***packs do not have cost updated***)*.
- If the recalculate order indicator is 'Y', update all relevant order/item/locations unit cost.
- If pack processing is necessary, repeat above steps except updating item location stock on hand.

Message DTD

Here are the filenames that correspond with the message type. Please consult the RIB documentation to get a detailed picture of the composition of the message.

Message Type	Message Type Description	DTD Name
Xcostchgmod	External Cost Change Modify	XCostChgDesc.dtd

Design Assumptions

- Required fields are shown in the RIB documentation.
- Updating the order cost does not take into account any aspects of building the order cost (estimated landed cost, deals, bracket cost, and so on) and will not work for a base solution.
- This API does not take into account estimated landed cost.
- This API assumes 'A'verage cost accounting. Hence no logic exists for 'S'tandard (last received) cost accounting.

Tables

TABLE	SELECT	INSERT	UPDATE	DELETE
ITEM_SUPP_COUNTRY	Yes	No	Yes	No
ITEM_SUPP_COUNTRY_LOC	Yes	No	Yes	No
ITEM_LOC_SOH	Yes	No	Yes	No
STORE	Yes	No	No	No
WH	Yes	No	No	No
ITEM_MASTER	Yes	No	No	No
DIFF_GROUP_HEAD	Yes	No	No	No
DIFF_GROUP_DETAIL	Yes	No	No	No
CHAIN	Yes	No	No	No
AREA	Yes	No	No	No
REGION	Yes	No	No	No
DISTRICT	Yes	No	No	No
ITEM_LOC	Yes	No	No	No
ORDLOC	Yes	No	Yes	No
ORDHEAD	Yes	No	No	No
PRICE_HIST	No	Yes	No	No
SYSTEM_OPTIONS	Yes	No	No	No

Currency Exchange Rates

Functional Area

Currency Exchange Rates

Business Overview

Currency exchange rates constitute financial information that is published to the Oracle Retail Integration Bus (RIB). A currency exchange rate is the price of one country's currency expressed in another country's currency.

RMS subscribes to currency exchange rates messages that are held on the RIB. The currency exchange rate message is a flat message that consists of a currency exchange rate record. The record contains information about the currency exchange rate as a whole. RMS places the information onto RMS tables depending upon the validity of the records enclosed within the message.

Note: When the RMS and the financial system are initially set up, identical currency information (3-letter codes, exchange rate values) is entered into both. If a new currency needs to be used, it must be entered into both the financial system and RMS before a rate change is possible. No functionality currently exists to bridge this data.

Data Flow

An external system will publish a currency exchange rate, thereby placing the currency exchange rate information onto the RIB. RMS will subscribe to the currency exchange rate information as published from the RIB and place the information onto RMS tables depending upon the validity of the records enclosed within the message.

Message Structure:

The currency exchange rate message is a flat message that will consist of a currency exchange rate record.

The record will contain information about the currency exchange rate as a whole.

Package Impact

Filename: rmssub_curratecres/b.pls

Subscribing to a currency exchange rate message entails the use of one public consume procedure. This procedure corresponds to the type of activity that can be done to currency exchange rate record (in this case create/update).

Public API Procedures:

```
RMSSUB_CURRATECRE.CONSUME
              (O_status_code        IN OUT        VARCHAR2,
               O_error_message      IN OUT        VARCHAR2,
               I_message            IN            CLOB)
```

This procedure accepts a XML file in the form of an Oracle CLOB data type from the RIB (I_message). This message will contain a currency exchange rate message consisting

of the aforementioned record. The procedure will then place a call to the main RMSSUB_CUR_RATES.CONSUME function in order to validate the XML file format and, if successful, parse the values within the file through a series of calls to RIB_XML. The values extracted from the file will then be passed on to private internal functions, which will validate the values and place them on the currency exchange rate table depending upon the success of the validation.

Private Internal Functions and Procedures (rmssub_curratecre.pls)

Error Handling:

If an error occurs in this procedure, a call will be placed to HANDLE_ERRORS in order to parse a complete error message and pass back a status to the RIB.

```
HANDLE_ERRORS
           (O_status                IN OUT         VARCHAR2,
            IO_error_message        IN OUT         VARCHAR2,
            I_cause                 IN             VARCHAR2,
            I_program               IN             VARCHAR2))
```

This function is used to put error handling in one place in order to make future error handling enhancements easier to implement. All error handling in the internal RMSSUB_CUR_RATES package and all errors that occur during subscription in the RMSSUB_CURRATECRE package (and whatever packages it calls) will flow through this function.

The function consists of a call to API_LIBRARY.HANDLE_ERRORS. API_LIBRARY.HANDLE_ERRORS accepts a program name, the cause of the error and potentially an unparsed error message if one has been created through a call to SQL_LIB.CREATE_MESSAGE. The function uses these input variables to parse a complete error message and pass back a status, depending upon the message and error type, back up through the consume function and up to the RIB.

Private Internal Functions and Procedures (other):

All of the following functions exist within RMSSUB_CUR_RATES.

Main Consume Function:

```
RMSSUB_CUR_RATES.CONSUME
               (O_error_message       OUT            VARCHAR2,
                I_message             IN             CLOB)
```

This procedure accepts a XML file in the form of an Oracle CLOB data type from the RIB (I_message) from the aforementioned public curratecre procedure whenever a message is made available by the RIB. This message will consist of the aforementioned record.

The procedure will then validate the XML file format and, if successful, parse the values within the file through a series of calls to RIB_XML. The values extracted from the file will then be passed on to private internal functions, which will validate the values and place them on the appropriate currency exchange rate database table depending upon the success of the validation.

XML Parsing:

PARSE_HEADER

This function will be used to extract the currency exchange rate level information from the currency exchange rate xml file and place that information onto an internal currency exchange rate record.

Validation:

PROCESS_HEADER

After the values are parsed for a particular currency exchange rate record, RMSSUB_CUR_RATES.CONSUME will call this function, which will in turn call various functions inside RMSSUB_CUR_RATES in order to validate the values and place them on the appropriate currency exchange rate table depending upon the success of the validation. CONVERT TYPE is called to validate the passed in currency rate if it exists in the FIF_CURRENCY_XREF table. PROCESS_RATES is called to actually insert or update the currency exchange rate table.

CONVERT_TYPE

This function would take in the current record's exchange rate type and return the RMS exchange type from the table FIF_CURRENCY_XREF. If no data was found, it should return an error message.

PROCESS_RATES

This function would call VALIDATE_RATES to ensure that the values passed from the message are valid. If all the values are valid, it would check if the currency code exists in the currency exchange rate table. If the currency code does not exist yet, call the function INTEREST RATES. If not, call UPDATE RATES.

VALIDATE_RATES

This function would pass each value from the record to the function CHECK_NULLS. CHECK_SYSTEM is used for conversion date.

CHECK_NULLS

This function would simply check if the values passed are NULL. If the passed value is NULL, then return an invalid parameter error message.

CHECK_SYSTEM

This function would fetch the vdate and the currency code from the period and system options table respectively. If the vdate is greater than the conversion date, return an error message. If the passed in currency rate is not the same as the currency rate fetched from the system options table, return an error message.

DML Module:

INSERT_RATES

This function would insert into the currency exchange rate table after all of the validations of the values are done.

UPDATE_RATES

This function would lock the CURRENCY_RATES table first. After that the table is locked. It would update the record in the currency exchange rate table.

Message DTD

Here are the filenames that correspond with each message type. Please consult the RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
CurrateCre	Currency Rate Create Message	CurrRateDesc.dtd
CurrateCre	Currency Rate Modify Message	CurrRateDesc.dtd

Design Assumptions

- One of the primary assumptions in the current API approach is that ease of code will outweigh performance considerations. It is hoped that the ‘trickle’ nature of the flow of data will decrease the need to dwell on performance issues and instead allow developers to code in the easiest and most straight forward manner.
- The adaptor is only setup to call stored procedures, not stored functions. Any public program then needs to be a procedure.

Tables

TABLE	SELECT	INSERT	UPDATE	DELETE
CURRENCY_RATES	Yes	Yes	Yes	No
SYSTEM_OPTIONS	Yes	No	No	No
PERIOD	Yes	No	No	No
FIF_CURRENCY_XREF	Yes	No	No	No

Diff Group

Functional Area

Diff Group

Design Overview

Differentiator subscriptions come into RMS from an external system. With a differentiator group subscription, you create the differentiator group in the external system, and then send that information to RMS. Once the subscription has been received, RMS users can now use the differentiator group that comes from the external system. The group is always sent first; its IDs are sent second.

System of record

The sor_item_ind system option indicates whether RMS is the system of record for differentiator (diff) groups. If RMS is the system of record, then RMS databases hold that information. If RMS is not the system of record, then that information is imported into RMS from outside systems. By setting the value of sor_item_ind to N (No), retailers no longer have the ability to create or edit diff groups online in RMS. Retailers can, however, continue to view diff groups.

When RMS is not the system of record for diff groups, the new diff group subscription API provides the data necessary to keep RMS in sync with an external system.

Differentiators

Differentiators augment RMS' item level structure by allowing you to define more discrete characteristics of an item. You attach differentiators to items to distinguish one item from another. Differentiators (diffs) give you the means to further track merchandise sales transactions. Common types of diffs are size, color, flavor, scent, or pattern.

Diffs consist of:

- Diff types – Generic categories of diff IDs such as Size, Color, or Flavor.
- Diff IDs – Specific attributes such as black, white, red; small, medium; strawberry, blueberry.
- Diff groups – Logical groupings of related diff IDs such as: Women's Pant Sizes, Shirt Colors, or Yogurt Flavors.

This API allows external systems to create, edit, and delete diff groups within RMS. The transaction will be performed immediately upon message receipt so success or failure can be communicated to the calling application.

Diff ID details can be created, edited, or deleted within the diff group message. Diff ID details must be created within a diff group on a diff group create message, they can also be passed in with their own specific message type. Diff ID detail create and modify messages will send a snapshot of the diff group record. Diff ID detail delete messages will be processed separately from the diff group delete because they have their own message types.

Consume Module

Filename: rmssub_xdiffgrps/b.pls

```
RMSSUB_XDIFFGRP.CONSUME
               (O_status_code        IN OUT        VARCHAR2,
                O_error_message      IN OUT        VARCHAR2,
                I_message            IN            RIB_OBJECT,
                I_message_type       IN            VARCHAR2)
```

This procedure will need to initially ensure that the passed in message type is a valid type for diff IDs messages. If the message type is invalid, a status of “E” should be returned to the external system along with an appropriate error message informing the external system that the status is invalid.

If the message type is valid, the generic RIB_OBJECT need to be downcast to the actual object using the Oracle’s treat function. If the downcast fails, a status of “E” should be returned to the external system along with an appropriate error message informing the external system that the object passed in is invalid.

If the downcast is successful, then consume needs to verify that the message passes all of the RMS business validation. It calls the RMSSUB_XDIFFGRP_VALIDATE.CHECK_MESSAGE function to determine whether the message is valid. If the message passed RMS business validation, then the function will return true; otherwise, it will return false. If the message has failed RMS business validation, a status of “E” should be returned to the external system along with the error message returned from the CHECK_MESSAGE function.

Once the message has passed RMS business validation, it can be persisted to the RMS database. It calls the RMSSUB_XDIFFGRP_SQL.PERSIST_MESSAGE() function. If the database persistence fails, the function will return false. A status of “E” should be returned to the external system along with the error message returned from the PERSIST_MESSAGE() function.

Once the message has been successfully persisted, there is nothing more for the consume procedure to do. A success, “S”, status should be returned to the external system indicating that the message has been successfully received and persisted to the RMS database.

RMSSUB_XDIFFGRP.HANDLE_ERROR() – This is the standard error handling function that wraps the API_LIBRARY.HANDLE_ERROR function.

Business Validation Module

It should be noted that some of the business validation is referential or involves uniqueness. This validation is handled automatically by the referential integrity constraints and the unique indexes implemented on the database.

Filename: rmssub_xdiffgrpvals/b.pls

```
RMSSUB_XDIFFGRP_VALIDATE.CHECK_MESSAGE
               (O_error_message       IN OUT         VARCHAR2,
                O_diffgroup_rec       OUT            DIFF_GROUP_REC,
                I_message             IN             RIB_XdiffgrpDesc,
                I_message_type        IN             VARCHAR2)
```

This function performs all business validation associated with the messages and builds the diff group record for persistence.

DIFF GROUP CREATE

- Check required fields.
- Verify diff group ID not used in diff ID table.
- Verify diff ID detail node is populated.
- Verify diff ID details are on diff ID table (not diff group table).
- Populate record with message data.

DIFF GROUP MODIFY

- Check required fields.
- Verify the diff group exists.
- Populate record with message data.

DIFF GROUP DELETE

- Check required fields.
- Verify the Diff group exists.
- Verify diff group is not attached to any items or pack templates.
- Populate record with message data.

DIFF ID CREATE

- Check required fields.
- Verify diff ID detail node is populated.
- Verify diff ID details are on diff ID table (not diff group table).
- Populate record with message data.

DIFF ID MODIFY

- Check required fields.
- Verify diff group exists.
- Verify diff ID detail node is populated.
- Verify diff ID details are on diff ID table (not diff group table).
- Verify diff ID details on diff group detail table.
- Populate record with message data.

DIFF ID DELETE

- Check required fields.
- Verify diff group exists.
- Verify the diff ID exists on diff group table.
- Verify no items or pack templates are using that diff group detail diff ID.
- Populate record with message data.

Bulk or single DML module

All insert, update and delete SQL statements are located in the family package. This package is DIFF_GROUP_SQL. The private functions will call this package.

Filename: rmssub_xdiffgrpsqls/b.pls

```
RMSSUB_XDIFFGRP_SQL.PERSIST_MESSAGE
               (O_error_message       IN OUT         VARCHAR2,
                I_diff_group_rec      IN             DIFF_GROUP_REC,
                I_message_type        IN             VARCHAR2,)
```

This function determines what type of database transaction it will call based on the message type.

DIFF GROUP CREATE

- Create messages get added to the Diff group head table.
- Diff group details get added to the diff group detail table.

DIFF GROUP MODIFY

- Modify messages directly update the Diff group head table with changes.

DIFF GROUP DELETE

- Delete messages directly remove Diff group head records.

DIFF GROUP DETAIL CREATE

- Create messages get added to the Diff group detail table.

DIFF GROUP DETAIL MODIFY

- Modify messages directly update the Diff group detail table with changes.

DIFF GROUP DETAIL DELETE

- Delete messages directly remove Diff group detail records.

Message DTD

Here are the filenames that correspond with each message type. Please consult RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Type	Message Type Description	Document Type Definition (DTD)
Xdiffgrpdtlcre	Create a diff group detail	XDiffGrpDesc.dtd
Xdiffgrpdtldel	Delete a diff group detail	XDiffGrpRef.dtd
xdiffgrpdtlmod	Modify a diff group detail	XDiffGrpDesc.dtd
xdiffgrpcre	Create a diff group header	XDiffGrpDesc.dtd
xdiffgrpdel	Delete an entire diff group	XDiffGrpRef.dtd
xdiffgrpmod	Modify a diff group header	XDiffGrpDesc.dtd

Design Assumptions

Required fields are shown in the RIB documentation.

Diff IDs and Diff groups must be validated for uniqueness, as they cannot overlap.

Tables

TABLE	SELECT	INSERT	UPDATE	DELETE
DIFF_IDS	Yes	No	No	No
DIFF_GROUP_HEAD	Yes	Yes	Yes	Yes
DIFF_GROUP_DETAIL	Yes	Yes	Yes	Yes
ITEM_MASTER	Yes	No	No	No
PACK_TMPL_HEAD	Yes	No	No	No
DIFF_RANGE_HEAD	Yes	No	No	No

Diff ID

Functional Area

Diff ID

Design Overview

When RMS is not the system of record for diff IDs, the diff ID subscription API provides the data necessary to keep RMS in sync with an external system.

The sor_item_ind system option indicates whether RMS is the system of record for differentiators (diffs). If RMS is the system of record, then RMS databases hold that information. If RMS is not the system of record, then that information is imported into RMS from outside systems. By setting the value of sor_item_ind to N (No), retailers no longer have the ability to create or edit diff IDs online in RMS. Retailers can, however, continue to view diff IDs.

This API allows external systems to create, edit, and delete Diff Ids within RMS. These transactions are performed immediately upon message receipt so success or failure can be communicated to the calling application.

For a general discussion of differentiators, see the section 'Diff Group' in the chapter "Subscription Design" in this volume of the RMS Operations Guide.

Consume Module

Filename: rmssub_xdiffids/b.pls

```
RMSSUB_XDIFFID.CONSUME
               (O_status_code        IN OUT        VARCHAR2,
                O_error_message      IN OUT        VARCHAR2,
                I_message            IN            RIB_OBJECT,
                I_message_type       IN            VARCHAR2)
```

This procedure will need to initially ensure that the passed in message type is a valid type for diff IDs messages. If the message type is invalid, a status of "E" should be returned to the external system along with an appropriate error message informing the external system that the status is invalid.

If the message type is valid, the generic RIB_OBJECT needs to be downcast to the actual object using the Oracle's treat function. If the downcast fails, a status of "E" should be returned to the external system along with an appropriate error message informing the external system that the object passed in is invalid.

If the downcast is successful, then consume needs to verify that the message passes all of the RMS business validation. It calls the RMSSUB_XDIFFID_VALIDATE.CHECK_MESSAGE function to determine whether the message is valid. If the message passed RMS business validation, then the function will return true; otherwise it will return false. If the message has failed RMS business validation, a status of "E" should be returned to the external system along with the error message returned from the CHECK_MESSAGE function.

Once the message has passed RMS business validation, it can be persisted to the RMS database. It calls the RMSSUB_XDIFFID_SQL.PERSIST_MESSAGE() function. If the database persistence fails, the function will return false. A status of "E" should be returned to the external system along with the error message returned from the PERSIST_MESSAGE() function.

Once the message has been successfully persisted, there is nothing more for the consume procedure to do. A success, "S", status should be returned to the external system indicating that the message has been successfully received and persisted to the RMS database.

RMSSUB_XDIFFID.HANDLE_ERROR() – This is the standard error handling function that wraps the API_LIBRARY.HANDLE_ERROR function.

Business Validation Module

It should be noted that some of the business validation is referential or involves uniqueness. This validation is handled automatically by the referential integrity constraints and the unique indexes implemented on the database.

Filename: rmssub_xdiffidvals/b.pls

```
RMSSUB_XDIFFID_VALIDATE.CHECK_MESSAGE
               (O_error_message       IN OUT         VARCHAR2,
                O_diffid_rec          OUT            DIFF_ID_REC,
                I_message             IN             RIB_XDiffIDDesc,
                I_message_type        IN             VARCHAR2)
```

This function performs all business validation associated with messages and builds the diff ID record for persistence.

DIFF ID CREATE

- Check required fields.
- Verify diff id not used in diff group head table.
- Populate record with message data.

DIFF ID MODIFY

- Check required fields.
- Verify the Diff Id exists.
- Populate record with message data.

DIFF ID DELETE

- Check required fields.
- Verify the Diff Id exists.
- Populate record with message data.

Bulk or single DML module

All insert, update and delete SQL statements are located in the family package. This package is DIFF_ID_SQL. The private functions will call this package.

Filename: rmssub_xdiffidsqls/b.pls

```
RMSSUB_XDIFFID_SQL.PERSIST_MESSAGE
               (O_error_message       IN OUT         VARCHAR2,
                I_diffid_rec          IN             DIFF_ID_REC,
                I_message_type        IN             VARCHAR2,)
```

This function determines what type of database transaction it will call based on the message type.

DIFF ID CREATE

- Create messages get added to the Diff ID table.

DIFF ID MODIFY

- Modify messages directly update the Diff ID table with changes.

DIFF ID DELETE

- Delete messages directly remove Diff ID records.

Message DTD

Here are the filenames that correspond with each message type. Please consult RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Type	Message Type Description	DTD Name
xdiffidcre	External Differentiator Create	XDiffIDDesc.dtd
xdiffiddel	External Differentiator Delete	XDiffIDRef.dtd
xdiffidmod	External Differentiator Modify	XDiffIDDesc.dtd

Design Assumptions

Required fields are shown in the RIB documentation.

Tables

TABLE	SELECT	INSERT	UPDATE	DELETE
DIFF_IDS	Yes	Yes	Yes	Yes
DIFF_GROUP_HEAD	Yes	No	No	No

Direct Ship Receipt

Functional Area

Direct Ship Receipt Subscription

Business Overview

In the direct ship receipt process, a retailer does not own inventory, but still records a sale on their books.

An external integration subsystem takes the order and sends it to a supplier.

When an integration subsystem is notified that a direct ship order is sent from the supplier, it publishes a new direct ship (DS) receipt message to the RIB for RMS' subscription purposes. RMS can then account for the data in the stock ledger.

Processing in conjunction with the subscription ensures that the weighted average cost for the item is recalculated.

RMS subscribes to integration subsystem direct ship receipt (DSR) messages. This records the inventory and financial transactions associated with the direct shipment of merchandise.

Package Impact

Filename: rmssub_dsrcpts/b.pls

```
RMSSUB_DSRCPT.CONSUME
               (O_status_code         IN OUT         VARCHAR2,
                O_error_message       IN OUT         RTK_ERRORS.RTK_TEXT%TYPE,
                I_message             IN             RIB_OBJECT,
                I_message_type        IN             VARCHAR2)
```

CONSUME simply calls different functions within the corresponding VALIDATE and SQL packages.

Before calling any functions, CONSUME narrows I_message down to the specific object being used, depending on the message_type. For example, a 'Cre' or 'Mod' message type usually means a 'Desc' object is being used. A 'Del' message usually means a 'Ref' object is being used. Object narrowing is done using the TREAT function. If the narrowing fails, then the CONSUME function should return an error message to the RIB stating that the object is not valid for this message family.

CONSUME first calls the family's VALIDATE package to validate the contents of the message. The family's SQL package is then called to perform DML.

Business Validation Module

Filename: rmssub_dsrcpt_vals/b.pls

```
CHECK_MESSAGE
          (O_error_message   IN OUT         RTK_ERRORS.RTK_TEXT%TYPE,
           O_dsrcpt_rec      OUT NOCOPY     RMSSUB_DSRCPT.DSRCPT_REC_TYPE,
           I_message         IN             RIB_XORDERDESC_REC,
           I_message_type    IN             VARCHAR2)
```

This function first calls the CHECK_FIELDS function to make sure all required fields are not NULL. Then, the function will call other functions as needed to validate all of the information that has been passed to it from the RIB.

DML Module

Filename: rmssub_dsrcpt_sqls/b.pls

```
RMSSUB_DSRCPT_SQL.PERSIST
               (O_error_message       IN OUT  RTK_ERRORS.RTK_TEXT%TYPE,
                I_dsrcpt_rec          IN      RMSSUB_DSRCPT.DSRCPT_REC_TYPE,
                I_message_type        IN      VARCHAR2)
```

This function will perform the inventory and financial transactions associated with the direct ship receipt. This includes updating the stock on hand and average cost for the item at the virtual store against which the direct shipment is being received, and, booking the associated purchase to the stock ledger for the item / virtual store.

Message DTD

Here are the filenames that correspond with each message type. Please consult RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
Dsrcptcre	Dsrcpt Create Message	DsrcptDesc.dtd

Design Assumptions

The subscriber makes some assumptions with the publisher's ability to maintain data integrity. The subscriber will not check for duplicate create messages. It will not check for missing messages because it has no way of knowing what would be missing. It also assumes that messages are sent in the correct sequence.

Tables

TABLE	SELECT	INSERT	UPDATE	DELETE
ITEM_MASTER	Yes	No	No	No
PACKITEM	Yes	No	No	No
ITEM_LOC_SOH	Yes	No	Yes	No
TRAN_DATA	No	Yes	No	No

DSD Deals

Functional Area

DSD deals subscription

Business Overview

Direct store delivery (DSD) is the delivery of merchandise and/or services to a store without the benefit of a pre-approved purchase order. DSD occurs when the supplier drops off merchandise directly in the retailer's store. This process is common in convenience and grocery stores, where suppliers routinely come to restock merchandise. In these cases, the invoice may or may not be given to the store (as opposed to sent to corporate), and the invoice may or may not be paid for out of the register.

RMS subscribes to DSD messages from the RIB. These messages notify RMS of a direct store delivery transaction at a location so that it may record the purchase order and account for it in the store’s inventory.

The receipt message that enters RMS includes information such as unit quantity, location, and so on. Based on the data, RMS performs the following functionality, as necessary.

- Creates a purchase order.
- Applies any deals.
- Creates a shipment.
- Receives a shipment.
- Creates an invoice.

Note: If ReIM is not running, invoices are not created.

The DSD consume package publishes a DSDDeals message much like receiving publishes an OTB message. This message is placed on a queue and is also processed by the DSDDeals consume package. The RIB has the capacity to process both the publishing result of the code in this package and the subscription to what the API publishes.

Package Impact

Filename: rmssub_dsddealss/b.pls

```
RMSSUB_DSDDEALS.CONSUME
               (O_status_code               IN OUT        VARCHAR2,
                O_error_message             IN OUT        VARCHAR2,
                I_rib_dsddealsdesc_rec      IN            RIB_DSDDealsDesc_REC,
                I_message_type              IN            VARCHAR2)
```

This procedure will need to initially ensure that the passed in message type is a valid type for DSD deals. The valid message type for DSD deals messages are listed in a section below.

If the message type is invalid, a status of “E” should be returned to the external system along with an appropriate error message informing the external system that the status is invalid.

Consume must call SYSTEM_OPTIONS_SQL.GET_SYSTEM_OPTIONS to get the required values for further processing.

For each header level data in the DSD deals table, call the function COMPLETE_TRANSACTION to persist data to the RMS database.

Once the message has been successfully persisted, there is nothing more for the consume procedure to do. A success status, “S”, is returned to the external system indicating that the message has been successfully received and persisted to the RMS database.

If an error occurs in this procedure, a call will be placed to HANDLE_ERRORS in order to parse a complete error message and pass back a status to the RIB.

RMSSUB_DSDDEALS.COMPLETE_TRANSACTION

This function checks for a shipment record on the shipment table for the DSD being processed. If no shipment record exists, it applies any applicable deals to the DSD order being processed and inserts shipment records into the shipment and shipsku tables for the newly created purchase order. After creating the new shipment, it receives the shipment and approves the order. If the DSD message contains invoice information, it creates the invoice.

```
RMSSUB_DSDDEALS.HANDLE_ERRORS
              (O_status             IN OUT        VARCHAR2,
               IO_error_message     IN OUT        VARCHAR2,
               I_cause              IN            VARCHAR2,
               I_program            IN            VARCHAR2)
```

This function is used to put error handling in one place in order to make future error handling enhancements easier to implement. All error handling in the internal RMSSUB_DSDDEALS package during subscription will flow through this function.

The function consists of a call to API_LIBRARY.HANDLE_ERRORS. API_LIBRARY.HANDLE_ERRORS accepts a program name, the cause of the error and potentially an unparsed error message if one has been created through a call to SQL_LIB.CREATE_MESSAGE. The function uses these input variables to parse a complete error message and pass back a status, depending upon the message and error type, back up through the consume function and up to the RIB.

Message DTD

Here are the filenames that correspond with each message type. Please see RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
dsddealscre	DSD Deals Create Message	DSDDealsDesc.dtd

Design Assumptions

None.

Tables

TABLE	SELECT	INSERT	UPDATE	DELETE
SHIPMENT	Yes	Yes	No	No
SHIPSKU	No	Yes	No	No
ORDAUTO_TEMP	Yes	No	No	Yes
ORDSKU	Yes	No	No	No
ORDLOC	Yes	No	No	No

DSD Receipt

Functional Area

DSD Receipt subscription

Business Overview

Direct store delivery (DSD) is the delivery of merchandise and/or services to a store without the benefit of a pre-approved purchase order. When the delivery occurs, the integration subsystem informs RMS of the receipt so a purchase order is created and it is counted in the store's inventory.

Package Impact

Filename: rmssub_dsds/b.pls

```
RMSSUB_DSD.CONSUME
               (O_status_code         IN OUT         VARCHAR2,
                O_error_message       IN OUT         VARCHAR2,
                I_rib_dsddesc_rec     IN             RIB_DSDReceiptDesc_REC,
                I_message_type        IN             VARCHAR2,
                O_rib_dsddeals_rec    OUT            RIB_DSDDealsDesc_REC)
```

The passed in message type should be a valid type for DSD receipts. The valid message type for DSD Receipts messages are listed in a section below.

If the message type is invalid, a status of "E" should be returned to the external system along with an appropriate error message informing the external system that the status is invalid.

If the message type is DSD_CRE, it performs validation on the values in the message. If the data is valid, it processes the "non-merch" data and detail level data before persisting the data to RMS databases.

If the message type is DSD_MOD, call the GET_ORDER_NO function to find the order number for the DSD.

If the message type is a create message, the O_rib_dsddeals_rec record is populated and passed back to the RIB so that it may be sent to the RMSSUB_DSDDEALS consume function. If the message type is not a create, then the O_rib_dsddeals_rec should be set to null.

Once the message has been successfully persisted, there is nothing more for the consume procedure to do. A success status, "S", is returned to the external system indicating that the message has been successfully received and persisted to the RMS database.

If an error occurs in this procedure, a call will be placed to HANDLE_ERRORS in order to parse a complete error message and pass back a status to the RIB.

RMSSUB_DSD.GET_ORDER_NO

This function is called whenever the message type is DSD_MOD. This function retrieves the current order number by searching the shipment tables using the external receipt number, store number and supplier.

```
RMSSUB_DSD.HANDLE_ERRORS
              (O_status             IN OUT        VARCHAR2,
               IO_error_message     IN OUT        VARCHAR2,
               I_cause              IN            VARCHAR2,
               I_program            IN            VARCHAR2)
```

This function is used to put error handling in one place in order to make future error handling enhancements easier to implement. All error handling in the internal RMSSUB_DSD package during subscription will flow through this function.

The function consists of a call to API_LIBRARY.HANDLE_ERRORS. API_LIBRARY.HANDLE_ERRORS accepts a program name, the cause of the error and potentially an unparsed error message if one has been created through a call to SQL_LIB.CREATE_MESSAGE. The function uses these input variables to parse a complete error message and pass back a status, depending upon the message and error type, back up through the consume function and up to the RIB.

Message DTD

Here are the filenames that correspond with each message type. Please consult the RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
dsdreceiptcre	DSD Receipt Create Message	DSDReceiptDesc.dtd
dsdreceiptmod	DSD Receipt Modify Message	DSDReceiptDesc.dtd

Design Assumptions

None.

Tables

TABLE	SELECT	INSERT	UPDATE	DELETE
SHIPMENT	Yes	No	No	No
ORDHEAD	Yes	No	No	No

Freight Terms

Functional Area

Freight Terms

Business Overview

Freight terms are financial arrangement information that is published to the Oracle Retail Integration Bus (RIB) from a financial system. Freight terms are the terms for shipping (for example, the freight terms could be a certain percentage of the total cost; a flat fee per order, etc.). RMS subscribes to freight terms messages held on the RIB. After confirming the validity of the records enclosed within the message, the RMS database is updated with the information.

Required fields in the message include a unique freight terms ID and a description.

Message Structure

The freight term message is a flat message that will consist of a freight term record.

Package Impact

Filename: rmssub_frttermcres/b.pls
rmssub_fterms/b.pls

Subscribing to a freight term message entails the use of one public consume procedure. This procedure corresponds to the type of activity that can be done to a freight term record (in this case create/update).

Public API Procedures

```
RMSSUB_FRTTERMCRE.CONSUME
               (O_status_code        IN OUT        VARCHAR2,
                O_error_message      IN OUT        VARCHAR2,
                I_message            IN            CLOB);
```

This procedure accepts a XML file in the form of an Oracle CLOB data type from the RIB (I_message). This message will contain a freight term message consisting of the aforementioned record. The procedure will then place a call to the main RMSSUB_FTERM.CONSUME function in order to validate the XML file format and, if successful, parse the values within the file through a series of calls to RIB_XML. The values extracted from the file will then be passed on to private internal functions, which will validate the values and place them on the freight term table depending upon the success of the validation.

Private Internal Functions and Procedures (rmssub_frttermcre.pls):

Error Handling

If an error occurs in this procedure, a call will be placed to HANDLE_ERRORS in order to parse a complete error message and pass back a status to the RIB.

```
HANDLE_ERRORS
           (O_status          IN OUT         VARCHAR2,
            IO_error_message  IN OUT         VARCHAR2,
            I_cause           IN             VARCHAR2,
            I_program         IN             VARCHAR2);
```

This function is used to put error handling in one place in order to make future error handling enhancements easier to implement. All error handling in the internal RMSSUB_FTERM package and all errors that occur during subscription in the RMSSUB_FRTTERMCRE package (and whatever packages it calls) will flow through this function.

The function consists of a call to API_LIBRARY.HANDLE_ERRORS. API_LIBRARY.HANDLE_ERRORS accepts a program name, the cause of the error and potentially an unparsed error message if one has been created through a call to SQL_LIB.CREATE_MESSAGE. The function uses these input variables to parse a complete error message and pass back a status, depending upon the message and error type, back up through the consume function and up to the RIB.

Private Internal Functions and Procedures (rmssub_fterm.pls):

All of the following functions exist within RMSSUB_FTERM.

Main Consume Function

```
RMSSUB_FTERM.CONSUME
              (O_status_code        IN OUT         VARCHAR2,
               O_error_message      IN OUT         VARCHAR2,
               I_message_clob       IN             CLOB)
```

This procedure accepts a XML file in the form of an Oracle CLOB data type from the RIB (I_message) from the aforementioned public rmssub_frttermcre procedure whenever a message is made available by the RIB. This message will consist of the aforementioned record.

The procedure will then validate the XML file format and, if successful, parse the values within the file through a series of calls to RIB_XML. The values extracted from the file will then be passed on to private internal functions, which will validate the values and place them on the appropriate freight term database table depending upon the success of the validation.

XML Parsing

PARSE_FTERM

This function will used to extract the freight term level information from the Freight Term XML file and place that information onto an internal freight term record.

Validation

PROCESS_FTERM

After the values are parsed for a particular freight term record, RMSSUB_FTERM.CONSUME will call this function, which will in turn call various functions inside RMSSUB_FTERM in order to validate the values and place them on the appropriate FREIGHT_TERMS table depending upon the success of the validation.

Message DTD

Here are the filenames that correspond with each message type. Please consult the RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
FrtTermCre	Freight Term Create Message	FrtTermDesc.dtd

Design Assumptions

- One of the primary assumptions in the current API approach is that ease of code will outweigh performance considerations. It is hoped that the 'trickle' nature of the flow of data will decrease the need to dwell on performance issues and instead allow developers to code in the easiest and most straight forward manner.
- The adaptor is only setup to call stored procedures, not stored functions. Any public program then needs to be a procedure.

Tables

TABLE	SELECT	INSERT	UPDATE	DELETE
FREIGHT_TERMS	Yes	Yes	Yes	No

GL Chart of Accounts

Functional Area

GL Chart of Accounts.

Business Overview

Before RMS can publish stock ledger data to an external financial application, it must receive that application's general ledger chart of accounts (GLCOA) structure. RMS accomplishes this through a subscription process.

A chart of account is essentially the financial application's debit and credit account segments (for example, company, cost center, account, and so on) that apply to the RMS product hierarchy. In some financial applications, this is known as code combination IDs (CCID). Upon receipt of GLCOA message data, RMS populates the data to the FIF_GL_ACCT table. The GL cross reference form is then used to associate the appropriate department, class, subclass, and location financial data to a chart that allows the population of that data to the GL_FIF_CROSS_REF table.

System Option for Financial Application

RMS' SYSTEM_OPTIONS table holds the column FINANCIAL_AP, where the interface financial application is indicated. Settings in this column are either 'O' or null. 'O' indicates an external financial application. A null indicates that no financial application is interfaced with RMS.

Data Flow

An external system will publish GL Chart of Accounts, thereby placing the GL chart of accounts information onto the Oracle Retail Integration Bus (RIB). RMS will subscribe to the GL chart of accounts information as published from the RIB and place the information onto RMS tables depending upon the validity of the records enclosed within the message.

Message Structure

The GL chart of accounts message is a flat message that will consist of a GL chart of accounts record.

The record will contain information about the GL chart of accounts as a whole.

Package Impact

Subscribing to a GL chart of accounts message entails the use of one public consume procedure. This procedure corresponds to the type of activity that can be done to currency exchange rate record (in this case create/update).

Public API Procedures:

Filename: rmssub_glcoacreb.pls

```
RMSSUB_ GLCOACRE.CONSUME
               (O_status_code        IN OUT        VARCHAR2,
                O_error_message      IN OUT        VARCHAR2,
                I_message            IN            CLOB)
```

This procedure accepts a XML file in the form of an Oracle CLOB data type from the RIB (I_message). This message will contain a GL chart of accounts message consisting of the aforementioned record. The procedure will then place a call to the main RMSSUB_GLCACCT.CONSUME function in order to validate the XML file format and, if successful, parse the values within the file through a series of calls to RIB_XML. The values extracted from the file will then be passed on to private internal functions, which will validate the values and place them on the GL chart of accounts table depending upon the success of the validation.

Private Internal Functions and Procedures (rmssub_glcoacreb.pls):

Error Handling:

If an error occurs in this procedure, a call will be placed to HANDLE_ERRORS in order to parse a complete error message and pass back a status to the RIB.

```
HANDLE_ERRORS
               (O_status             IN OUT        VARCHAR2,
                IO_error_message     IN OUT        VARCHAR2,
                I_cause              IN            VARCHAR2,
                I_program            IN            VARCHAR2)
```

This function is used to put error handling in one place in order to make future error handling enhancements easier to implement. All error handling in the internal RMSSUB_GLCACCT package and all errors that occur during subscription in the RMSSUB_GLCOACRE package (and whatever packages it calls) will flow through this function.

The function consists of a call to API_LIBRARY.HANDLE_ERRORS. API_LIBRARY.HANDLE_ERRORS accepts a program name, the cause of the error and potentially an unparsed error message if one has been created through a call to SQL_LIB.CREATE_MESSAGE. The function uses these input variables to parse a complete error message and pass back a status, depending upon the message and error type, back up through the consume function and up to the RIB.

Private Internal Functions and Procedures (other):

Filename: rmssub_glcacctb.pls

Main Consume Function:

```
RMSSUB_GLCACCT.CONSUME
                (O_ERROR_MESSAGE        OUT     VARCHAR2,
                 I_MESSAGE              IN      CLOB)
```

This procedure accepts a XML file in the form of an Oracle CLOB data type from the RIB (I_message) from the public rmssub_glcoacre.consume procedure whenever a message is made available by the RIB. This message will consist of the aforementioned record.

The procedure will then validate the XML file format and, if successful, parse the values within the file through a series of calls to RIB_XML. The values extracted from the file will then be passed on to private internal functions, which will validate the values and place them on the appropriate GL chart of accounts database table depending upon the success of the validation.

XML Parsing:

```
PARSE_HEADER
            (O_ERROR_MESSAGE    OUT             VARCHAR2,
             O_GLACCT_RECORD    OUT             GLACCT_RECTYPE,
             I_GLACCT_ROOT      IN OUT          xmldom.DOMElement)
```

This function extracts the GL chart of accounts level information from the GL Chart of Accounts XML file and places that information onto an internal GL Chart of Accounts record.

Record is based upon the record type glacct_rectype.

Validation:

PROCESS_HEADER

After the values are parsed for a particular GL chart of accounts record, RMSSUB_GLCACCT.CONSUME will call this function, which will in turn call various functions inside RMSSUB_GLCACCT in order to validate the values and place them on the appropriate GL chart of accounts table depending upon the success of the validation. PROCESS_GLACCT is called to actually insert or update the GL chart of accounts table.

PROCESS_GLACCT

Function PROCESS_GLACCT will take the input GL record and place the information into a local GL record which will be used in the package to manipulate the data. It will then call a series of support functions to perform all business logic on the record.

INSERT_GLACCT

Function INSERT_GLACCT will insert any valid account on the GL table. It is called from PROCESS_GLACCT.

UPDATE_GLACCT

Function UPDATE_GLACCT will insert any valid account on the GL table. It is called from PROCESS_GLACCT.

VALIDATE_GLACCT

Function VALIDATE_GLACCT is a wrapper function which is used to call CHECK_NULLS, CHECK_ATTRS for any GL record input into the package.

FUNCTION CHECK_NULLS

Function CHECK_NULLS will check an input value if it is null. If so, an error message will be created.

CHECK_ATTRS

Function CHK_ATTRS that is called within the validation function of this package to ensure that RMS will not accept incomplete data from a financial interface when sent via the RIB. This function will check to ensure that each description that is input also has an attribute that it describes.

Message DTD

Here are the filenames that correspond with each message type. Please consult the RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
Glcoacre	Glco Create Message	GLCOADesc.dtd

Design Assumptions

Required fields are shown in the RIB documentation.

Many ordering functionalities that are available on-line are not supported via this API. Triggers related to these functionalities should be turned off.

Tables

TABLE	SELECT	INSERT	UPDATE	DELETE
FIF_GL_ACCT	Yes	Yes	Yes	No

Inventory Adjustment

Functional Area

Inventory Adjustment

Business Overview

RMS receives requests for inventory adjustments from an integration subsystem through the inventory adjustment subscription. The requests contain information about the item, the physical warehouse, the quantity, the specific disposition change, and the reason for the adjustment. RMS uses data in these requests to:

- Adjust overall quantities of stock on hand for an item at a location.
- Adjust the availability of item-location quantities. Currently RMS interprets all stock dispositions contained in a message as available or unavailable.

After initial processing from the integration subsystem RMS performs the following tasks:

- Validates the item-location combinations and adjustment reasons.
- Updates stock on hand data for the item at the location.
- Inserts stock adjustment transaction codes on the RMS stock ledger.
- Adjusts quantities by inventory status for item/location combination.
- Create an audit trail for the inventory adjustment by item, location, inventory status and reason.

Inventory Quantity and Status Evaluation

RMS evaluates inventory adjustments to decide if overall item-location quantities have changed, or if the statuses of quantities have changed.

The FROM_DISPOSITION and TO_DISPOSITION tags in the message are evaluated to determine if there is a change in overall quantities of an item at a location. For the given item and quantity reported in the message, if either tag contains a null value, RMS evaluates that fact as a change in overall quantity in inventory.

In addition, if the message shows a change to the status of existing inventory, RMS evaluates this to determine if that change makes a quantity of an item unavailable.

Stock Adjustment Transaction Codes

Whenever the status or quantity of stock changes, RMS writes transaction codes to adjust inventory values in the stock ledger. The two types of inventory adjustment transaction codes are:

- Adjustments to total stock on hand, where positive and negative adjustments are made to total stock on hand. In this case, a ‘Stock Adjustment’ transaction (TRAN_CODE = ‘22’) is inserted on the Stock Ledger (TRAN_DATA table) for both retail and cost methods of accounting.
- Adjustments to unavailable (non-sellable) inventory. In this case, an ‘Unavailable Inventory Transfer’ transaction (TRAN_CODE = ‘25’) is inserted on the Stock Ledger (TRAN_DATA table).

Subscription Package

Filename: rmssub_invadjusts/b.pls

```
RMSSUB_INVADJUST.CONSUME
               (O_status_code        IN OUT        VARCHAR2,
                O_error_message      IN OUT        RTK_ERRORS.RTK_TEXT%TYPE,
                I_message            IN            RIB_OBJECT,
                I_message_type       IN            VARCHAR2)
```

This procedure will need to initially ensure that the passed in message type is a valid type for inventory adjustment messages. The valid message type for an inventory adjustment message is listed in a section below.

If the message type is invalid, a status of “E” should be returned to the external system along with an appropriate error message informing the external system that the status is invalid.

If the message type is valid, the generic RIB_OBJECT needs to be downcast to the actual object using the Oracle’s treat function. There will be an object type that corresponds with each message type. If the downcast fails, a status of “E” is returned to the external system along with an appropriate error message informing the external system that the object passed in is invalid.

If the downcast is successful, then consume needs to verify that the message passes all of the RMS business validation. It calls the INVADJ_SQL function to perform validation and to insert or update records in the database whether the message is valid. If the message passed RMS business validation and is successfully persisted in the database then a successful status is returned to the CONSUME. If the message fails RMS business validation or encounters any other errors, a status of “E” is returned to the external system along with the error message.

```
RMSSUB_INVADJUST.PROCESS_INVADJ
               (O_error_message      IN OUT        VARCHAR2,
                I_message            IN            RIB_InvAdjustDesc_REC)
```

This function calls CHECK_ITEMS, an internal function that checks for any sellable only “break to sell” items and separates these items into an object table for further processing. A table of the corresponding orderable items and quantities for the sellable items is built to submit to the inventory adjustment process.
INVADJ_SQL.PROCESS_INVADJ is called for the table of regular items and the table of “break to sell” items to perform all business validation and desired functionality associated with an inventory adjustment message.

Business Validation and DML Module

Filename: invadjs/b.pls

INVADJ_SQL.PROCESS_INVADJ

This function performs business validation and desired functionality for an inventory adjustment message. It includes the following:

- Check required fields: item, location, adj_qty, user_id, adj_date.
- Check valid values: The location type must be either ‘S’ or ‘W’. The doc_type must be NULL, ‘P’, or ‘T’.
- Verify that the to_disposition or from_disposition or both fields are populated. Both cannot be NULL.
- Verify that an orderable but non-sellable and non-inventory item **cannot** be an inventory adjustment item.
- If the item is a simple pack catch weight item, verify that weight and weight UOM are either both defined or both NULL, and weight UOM is in the MASS UOM class.
- Verify that the item is a tran-level or above tran-level item.
- Verify that the item/loc relation exists and create it if it does not exist.
- If adjusting a pack at a warehouse, the pack must be received as pack at the warehouse.
- Verify that from disposition is a valid inventory status code (on INV_STATUS_CODES).
- If multi-channel is on and the location is a warehouse, then physical location is on the message. The adjusted quantity is distributed among the virtual locations of the physical location.
- For available stock on hand the items are added to the update records for updating the ITEM_LOC_SOH table and a tran code 22or 25 data is prepared for writing the TRAN_DATA records. For external finisher location type and for transformable orderable items, the unit_retail is calculated with the appropriate package call for these two exception cases. The following additional processing for packs:
    - Packs that are received as packs have the pack quantity calculated. If the pack is a catch weight simple pack, the average weight is calculated to obtain the quantity.
    - For other packs, the updated quantity is set for each pack component. If the pack is a catch weight simple pack, the computed quantity is based on the input weight or the average weight.
    - The total cost is calculated before writing records to TRAN_DATA. For catch weight items the cost calculation is based on weight.
- If the I_cogs_ind input parameter is ‘Y’, the tran_code is changed to 23.
- For unavailable stock on hand, the unavailable quantities are computed before the items or the pack components are added to the update records for updating the ITEM_LOC_SOH table and a tran code 22 or 25 data is prepared for writing the TRAN_DATA records. For external finisher location type and for transformable orderable items, the unit_retail is calculated with the appropriate package call for these two exception cases.

Message DTD

Here are the filenames that correspond with each message type. Please consult the RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
invadjustcre	Inventory Adjustment Create Message	InvAdjustDesc.dtd

Design Assumptions

- One of the primary assumptions in the current API approach is that ease of code will outweigh performance considerations. It is hoped that the 'trickle' nature of the flow of data will decrease the need to dwell on performance issues and instead allow developers to code in the easiest and most straightforward manner.
- The adaptor is only setup to call stored procedures, not stored functions. Any public program then needs to be a procedure.

Tables

TABLE	SELECT	INSERT	UPDATE	DELETE
ITEM_LOC_SOH	Yes	Yes	Yes	No
TRAN_DATA(VIEW)	No	Yes	No	No
INV_ADJ	No	Yes	No	No
INV_STATUS_QTY	No	Yes	Yes	Yes
INV_ADJ_REASON	Yes	No	No	No
V_PACKSKU_QTY	Yes	No	No	No
ITEM_LOC	Yes	No	No	No
ITEM_MASTER	Yes	No	No	No
INV_STATUS_CODES	Yes	No	No	No
TSFHEAD	Yes	No	No	No
SHIPSKU	Yes	No	No	No

Inventory Request

Functional Area

Inventory Request Subscription

Business Overview

RMS receives requests for inventory from an integration subsystem through the inventory request subscription.

Store ordering allows for all items to be ordered by the store and fulfilled by an RMS process. RMS fulfills a store's request regardless of replenishment review cycles, delivery dates, and any other factors that may restrict a request from being fulfilled. However, delivery cannot always be guaranteed on or before the store's requested due date, due to supplier or warehouse lead times and other supply chain factors that may restrict on-time delivery.

Store ordering is used to request inventory for any items that are on the 'Store Order' type of replenishment. The store order replenishment process requires the store to request a quantity and builds the recommended order quantity (ROQ) based on the store's requests. Requests for store order items that will not be reviewed prior to the date requested by the store are fulfilled through a one-off process (run as needed) that creates warehouse transfers and/or purchase orders to fulfill the requested quantities.

Orders and transfers will be used to create and approve store orders to the supplier or the warehouse when the user notices there is a shortage or demand for particular items. The user will select either a warehouse or a supplier to create the order and add the items and the quantities. The order will then use an API to call RMS packages and validate all data is correct before creating and or approving an order from any integration subsystem. If the store order is for a warehouse, RMS will create a transfer from the warehouse to the store. Note the following system abilities and assumptions:

- Ability to create orders to the supplier or the warehouse.
- Ability to save the creation of the order without approving it.
- Ability to amend orders/transfers for items in integration subsystems that were created either manually or through replenishment in RMS.
- Requests with a request type of item request (IR) will generate transfers or orders if the item is not set up with replenishment, or the need date is before the next replenishment review date.
- Requests with a request type of store order (SO) must have a store order replenishment set up for the items. Transfers or orders will be generated if the need date is before the next replenishment review date.

Package Impact

Filename: rmssub_invreqs/b.pls

```
RMSSUB_INVREQ.CONSUME
              (O_status_code        IN OUT        VARCHAR2,
               O_error_message      IN OUT        VARCHAR2,
               I_message            IN            RIB_OBJECT,
               I_message_type       IN            VARCHAR2,
               O_rib_error_tbl      OUT           RIB_ERROR_TBL)
```

This procedure should initially downcast the generic RIB_OBJECT to the actual object using the Oracle's treat function.

If the downcast is successful, it should empty out the cache of inserts and updates to the store_orders table and to the PL/SQL ITEM_TBL table. This is done by calling INV_REQUEST_SQL.INIT function. Global variables to be used are going to be initialized by the function RMSSUB_INVREQ_ERROR.INIT. This should be called before processing any item/store order request.

Input from the header level info should be validated. If any of the required header level info is NULL, the entire request should be rejected; however, there is no need to write to the error table.

Once the header level info has passed validation, RMSSUB_INVREQ_ERROR.BEGIN_INVREQ should be called to hold the header level values into global variables which may be used to build an error record when necessary. Process each item by calling INV_REQUEST_SQL.PROCESS.

The cache for the STORE_ORDERS table and the PL/SQL ITEM_TBL table should be populated by calling INV_REQUEST_SQL.FLUSH function. At the end of the inventory request process, RMSSUB_INVREQ_ERROR.FINISH function should be called to pass a copy of the global error table (if any error exists) which is sent to the RIB for further processing.

Filename: rmssub_invreq_errors/b.pls

Most of the functions included are called by the RMSSUB_INVREQ.CONSUME procedure to process inventory requests.

```
RMSSUB_INVREQ_ERROR.INIT
              (O_error_message      IN OUT        RTK_ERRORS.RTK_TEXT%TYPE,
               I_message_type       IN            VARCHAR2)
```

This function would simply initialize all of the global variables which include the RIB_OBJECTS that are used to process the inventory request.

```
RMSSUB_INVREQ_ERROR.BEGIN_INVREQ
              (O_error_message      IN OUT        RTK_ERRORS.RTK_TEXT%TYPE,
               I_request_id         IN            NUMBER,
               I_store              IN            STORE_ORDERS.STORE%TYPE,
               I_request_type       IN            VARCHAR2)
```

This function would populate the global variables using the header level values to create an error record whenever necessary.

```
RMSSUB_INVREQ_ERROR.ADD_ERROR
               (O_error_message       IN OUT         RTK_ERRORS.RTK_TEXT%TYPE,
                I_error_desc          IN             VARCHAR2,
                I_error_object        IN             RIB_OBJECT)
```

This function would be called whenever an error is encountered during the processing of the inventory request. It adds the error type/description and error object on the global error table.

RMSSUB_INVREQ_ERROR.FINISH

This function is called after processing the inventory request. It passes out a copy of the global error table (if any error is present) to the RIB for further processing.

RMSSUB_INVREQ_ERROR.GET_MESSAGE_KEY

This function would get the key from a SQL_LIB error message. If the error message is just text without any parameters, the entire message will be passed back out as the key.

Filename: invrequests/b.pls

```
INV_REQUEST_SQL.PROCESS
           (O_error_message   IN OUT         VARCHAR2,
            I_store           IN             STORE_ORDERS.STORE%TYPE,
            I_request_type    IN             VARCHAR2,
            I_item            IN             STORE_ORDERS.ITEM%TYPE,
            I_need_qty        IN             STORE_ORDERS.NEED_QTY%TYPE,
            I_uop             IN             UOM_CLASS.UOM%TYPE,
            I_need_date       IN             STORE_ORDERS.NEED_DATE%TYPE)
```

This function does all the validation and processing of the inventory request. It creates a record for STORE_ORDERS or LP_ITEM_TBL (PL/SQL table for adhoc requests).

INV_REQUEST_SQL.VERIFY_REPL_INFO (local)

This function retrieves the replenishment information. If the request type is ‘IR’ and the item is not set up on replenishment, set adhoc to ‘Y’. Item requests with request type of ‘SO’ or NULL must have store order replenishment set up in RMS for that item. The need date must be after the next replenishment delivery date if the store order has been rejected by replenishment. If the need date is before the next replenishment review date for both request types, set adhoc to ‘Y’.

INV_REQUEST_SQL.FUNCTION CONVERT_NEED_QTY (local)

This function converts the need quantity to ‘E’aches for Packs.

INV_REQUEST_SQL.PREPARE_AD_HOC (local)

This function is called if the Adhoc indicator is set to ‘Y’. It will write the request to the PL/SQL table that will be passed to the function call CREATE_ORD_TSF_SQL.CREATE_ORD_TSF to create an order or transfer.

INV_REQUEST_SQL.VERIFY_ON_STORE (local)

This function checks to see if the item request already exists on STORE_ORDER. If it exists, call PREPARE_UPDATE to update the need quantity to include the new need quantity. If it does not, call PREPARE_INSERT to insert into STORE_ORDER table.

INV_REQUEST_SQL. PREPARE_INSERT (local)

This function checks the PL/SQL table that contains the BULK INSERT records. If a record exists on the PL/SQL table, update the qty.

INV_REQUEST_SQL. PREPARE_UPDATE (local)

This function adds a record to the PL/SQL table that contains the BULK UPDATE records.

INV_REQUEST_SQL. FLUSH(local)

This function does the actual insert or update to STORE_ORDERS.

Message DTD

Here are the filenames that correspond with each message type. Please consult the RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
InvReqCre	Inventory Request Create Message	InvReqDesc.dtd

Design Assumptions

RMS will explode a pack to its component items when the request is for a complex buyer pack requiring ordering as ‘Eaches’, from a supplier.

- RMS action: Explode pack to component level when the pack is a complex buyer pack ordered as ‘Eaches’.

RMS will explode a complex pack to its component items when the order is to a warehouse for a complex pack being received at the ordered-from warehouse as ‘Eaches’.

- RMS action: Explode pack to component level when the requested-from warehouse is receiving a pack as ‘Eaches’ and return the components.

RMS will round quantities using the store order multiple when an order is created for a warehouse.

- RMS action: Determine the store order multiple, and round the quantities accordingly. Return the rounded values to the integration subsystem.

RMS will explode a pack item to its component items when the request is for a pack that is received at the requested from warehouse as ‘Eaches’.

- RMS action: Determine receive as type for from Wh, and explode pack to component level.

Upcharges will always be applied to a transfer when they can be defaulted.

- RMS action: Always default upcharges for warehouse orders.

Soft warnings within RMS (for example, item is on clearance, OTB exceeded, and so on) that result in a ‘Yes/No’ option to the user when approving an order will always be assumed ‘Yes’, so as not to restrict an order from being built or approved within RMS.

- RMS action: Perform checks and return error string to the integration subsystem but not allow the errors to restrict processing (for example, creating order, editing, approving, and so on).

RMS should accept updates for both worksheet and submitted orders/transfers.

- RMS action: Transfers reserve inventory upon submittal of the transfer. When a 'submitted' transfer is edited, the reserved inventory must be updated accordingly. When a 'submitted' PO is edited, it should be set back to 'Worksheet' status automatically.

Another application might be able to override an item's cost when creating an order for a supplier.

- RMS action: Create the item/location record with a 'Manual' cost and do not apply deals to this item.
- Other application's action: Allow user to edit the unit cost and communicate the edited cost to RMS.

RMS will map a physical warehouse location to the appropriate virtual warehouse for another application's created orders to a warehouse.

- RMS action: When building a transfer, checking inventory, and so on, RMS should map the physical warehouse communicated by another application to the virtual warehouse within the passed physical location that contains the same channel ID as the store. The store will not have access to any inventory other than what is in this channel.
- Other application's action: Orders for a warehouse will be created and communicated using physical warehouses.

RMS will validate that all items belong to the same department when department level ordering (supplier) or department level transfers (Wh) are being used.

- RMS action: For an order to a supplier, if the department level orders indicator is 'Y', validate that all items belong to the same department. For an order to a warehouse, if the department level transfers indicator is 'Y', validate that all items belong to the same department.

RMS will validate that an item is not a consignment item if the order is for a warehouse.

- RMS action: Validate that each item on the order is not a consignment item. If a consignment item is found, return an error.

Another application will validate that at least one item exists on an order before it can be approved.

- RMS action: None
- Other application's action: Require the user to enter at least one item on an order before it can be approved.

RMS will validate that a store is open when the store is being transferred to.

- RMS action: Before a PO or transfer is approved; verify that the store on the order is open. If the order is to a supplier, verify that the store will be open on the not before date.
- RMS action: Users in RMS will have ability to manually split/scale orders, but no batch process will drive this.

Another application that generates its orders to default import ind to 'N'o

- RMS action: None. Landed cost will not apply to these orders; users will not be able to edit this indicator once the order is created.

Tables

TABLE	SELECT	INSERT	UPDATE	DELETE
STORE_ORDERS	Yes	Yes	Yes	No
REPL_ITEM_LOC	Yes	No	No	No
ITEM_LOC	Yes	No	No	No
ITEM_SUPP_COUNTRY	Yes	No	No	No
ITEM_MASTER	Yes	No	No	No
SUPS	Yes	No	No	No
ITEM_LOC_SOH	Yes	No	No	No
TSFHEAD	No	Yes	Yes	No
TSFDETAIL	Yes	Yes	Yes	No
ORDHEAD	No	No	Yes	No

Item

Functional Area

Item

Design Overview

The system option 'sor_item_ind' indicates whether RMS is the system of record for item maintenance. If RMS is not the system of record, then this API is used to import the data from an external system, and the data can be viewed in the RMS windows, but not created or modified.

Item/location relationships are not handled by this API; they are handled in a separate Item Location Subscription API.

When this API accepts messages with create message types, it inserts the data into the ITEM_MASTER, PACKITEM (in the case of a pack), ITEM_SUPPLIER, ITEM_SUPP_COUNTRY. Additionally, records can be inserted into the ITEM_SUPPLIER_COUNTRY_LOC table. The VAT_ITEM table is populated with data defaulted from the item's department. Optionally, the records can be inserted into the VAT_ITEM table to override these defaults. The messages with modify message types consist of snapshots of records for updating the ITEM_MASTER, ITEM_SUPPLIER, ITEM_SUPP_COUNTRY, ITEM_SUPPLIER_COUNTRY_LOC and VAT_ITEM tables.

Item messages include the required detail nodes for the supplier and supplier/country. If the item is not a non-sellable pack, the item/zone/price node is also required. Optional nodes can be included in the message for supplier/country/locations, pack components, and item/vat relationships.

Items must be created and maintained following a logical hierarchy as outlined by the referential integrity of the item database tables: Item parents before child items; item components before items that are packs; items before item-suppliers; item/suppliers before item/supplier/countries; items before item/locations (a separate API), and so on. Failing to do so, results in message failure.

The create and modify messages are hierarchical with required detail nodes of suppliers and supplier/countries and optional nodes for price zones, supplier/country/locations and vat codes. If the item is a pack item, the pack component node is required. In the header modify message, the detail nodes are not populated, but the full header node is sent. The detail level create or modify messages contains the item header record and one to many detail records in the node or nodes. For example, the message type of XItemSupMod could have one or more supplier details to update in the ITEM_SUPPLIER table. The modify messages contain a snapshot of the record for update rather than only the fields to be changed.

The auto-creation of item children using differentiator records attached to an item parent, as currently occurs using RMS online processes, is not supported in this API.

The delete messages contain only the primary key field for the item, supplier, supplier/country, supplier/country/location or vat/item record that is to be deleted. When a delete message is processed, the item is not immediately deleted; rather, it is added to the daily purge table. Deleting the item is a batch process.

Consume Module

Filename: rmssub_items/b.pls

```
RMSSUB_XITEM.CONSUME
                     (O_status_code         IN OUT VARCHAR2,
                      O_error_message       IN OUT VARCHAR2,
                      I_message             IN     RIB_OBJECT,
                      I_message_type        IN     VARCHAR2)
```

This procedure will need to initially ensure that the passed in message type is a valid type for organizational hierarchy messages. The valid message types for organizational hierarchy messages are listed in a section below.

If the message type is invalid, a status of "E" should be returned to the external system along with an appropriate error message informing the external system that the status is invalid.

If the message type is valid, the generic RIB_OBJECT needs to be downcast to the actual object using the Oracle's treat function. There will be an object type that corresponds with each message type. If the downcast fails, a status of "E" is returned to the external system along with an appropriate error message informing the external system that the object passed in is invalid.

If the downcast is successful, then consume needs to verify that the message passes all of the RMS business validation. It calls the RMSSUB_ XITEM_VALIDATE.CHECK_MESSAGE function to determine whether the message is valid. If the message passed RMS business validation, then the function returns true, otherwise it returns false. If the message fails RMS business validation, a status of "E" is returned to the external system along with the error message returned from the CHECK_MESSAGE function.

Once the message has passed RMS business validation, it is persisted to the RMS database. It calls the RMSSUB_XITEM_SQL.PERSIST_MESSAGE() function. If the database persistence fails, the function returns false. A status of "E" is returned to the external system along with the error message returned from the PERSIST_MESSAGE() function.

Once the message has been successfully persisted, there is nothing more for the consume procedure to do. A success status, "S", is returned to the external system indicating that the message has been successfully received and persisted to the RMS database.

RMSSUB_ITEM.HANDLE_ERROR () – This is the standard error handling function that wraps the API_LIBRARY.HANDLE_ERROR function.

Business Validation Module

Filename: rmssub_xitemvals/b.pls

It should be noted that some of the business validation is referential or involves uniqueness. This validation is handled automatically by the referential integrity constraints and the unique indexes implemented on the database.

```
RMSSUB_XITEM_VALIDATE.CHECK_MESSAGE
               (O_error_message       IN OUT         VARCHAR2,
                O_item_rec            OUT            NOCOPY
                RMSSUB_ITEM.ITEM_API_REC,
                I_message             IN             RIB_XItemDesc,
                I_message_type        IN             VARCHAR2)
```

This overloaded function performs all business validation associated with create/modify messages and builds the item API record with default values and locations from the organizational hierarchy for persistence in the item related tables. Any invalid records passed at any time results in message failure.

Defaulted fields that are not included in the message structure of the object must be populated in a package business record, ITEM_API_REC. This record is used as input to the database DML functions in the persist package.

ITEM_MASTER VALIDATION

CREATE

- Check required fields and nodes.
- Diff Ids exist, no duplicates, no nulls in order, part of parent diff group if item is a child item.
- Cost group zone is null for landed cost option, sellable only packs, buyer packs.
- Check valid values for item_level, tran_level, pack_type, store_ord_mult, and all indicators fields .
- Check organization hierarchy level for item_supp_country_loc and item pricing.
- Process all validations for item_supplier, item_supp_country, item pricing, and if nodes exist, packitem, item_supp_country_loc, vat_item.

MODIFY

- Check required fields.
- Check existence.

DELETE

- Check for primary key.
- Check existence of foreign keys.

ITEM_SUPPLIER VALIDATION

CREATE

- Check required fields.
- Check primary supplier, only 1 and 1 required.

MODIFY

- Check required fields.
- Check primary supplier, only 1 and 1 required.

DELETE

- Check for primary key.
- Check primary supplier, only 1 and 1 required.
- Check that the item/supplier is not part of the following:
  - On an order.
  - On an open return to vendor record.
  - On an open contract.
  - The specified item is not a component item to a pack item that is sourced from the specified supplier.
  - The specified supplier is not a primary supplier for any location to which the specified item has an existing relationship.
  - The specified supplier is not a primary replenishment supplier for the specified item.

ITEM_SUPP_COUNTRY VALIDATION

CREATE

- Check required fields
- Check primary country, only 1 and 1 required
- Unit cost 0 for buyer packs

MODIFY

- Check required fields
- Check primary country, only 1 and 1 required

DELETE

- Check for primary key
- Check primary country, only 1 and 1 required
  - Check that the item/supplier/country is not part of any of the following:
  - On an order.
  - The specified item is not a component item to a pack item that is sourced from the specified supplier/country.
  - The specified supplier/country is not a primary supplier/primary country for any location to which the specified item has an existing relationship.
  - The specified supplier/country is not a primary replenishment supplier/country combination for the specified item.
  - The specified country is not a primary country for the specified item and any supplier.

ITEM_SUPP_COUNTRY_LOC VALIDATION

CREATE

- Check required fields.
- Check primary location, only 1 and 1 required.
- Unit cost 0 for buyer packs.

MODIFY

- Check required fields.
- Check primary location, only 1 and 1 required.

DELETE

- Check for primary key.
- Check primary location, only 1 and 1 required.

PACKITEM VALIDATION

CREATE

- Check required fields.
- Check component items exist.

VAT_ITEM VALIDATION

CREATE

- Check required fields.
- Check valid values for VAT_TYPE.

DELETE

- Check for primary key.
- Check dept/class VAT_REGION association.

POPULATE ITEM_API_RECORD

ITEM_MASTER

CREATE

- Populate ITEM_MASTER defaults.
- Populate ITEM_SUPPLIER defaults.
- Populate ITEM_SUPP_COUNTRY defaults.
- Populate ITEM_SUPP_COUNTRY_LOC defaults.
- Populate ITEM_SUPP_COUNTRY_LOC table of location records.
- Populate IZP_TBL table of location records.

ITEM_SUPPLIER

CREATE

- Populate ITEM_SUPPLIER defaults.

MODIFY

- Populate ITEM_SUPPLIER mod table with existing value for primary_supp_ind.

ITEM_SUPP_COUNTRY

CREATE

- Populate ITEM_SUPP_COUNTRY defaults.

MODIFY

- Populate ITEM_SUPP_COUNTRY mod table with existing value for primary_country_ind.

ITEM_SUPP_COUNTRY_LOC

CREATE

- Populate ITEM_SUPP_COUNTRY_LOC defaults.
- Populate ITEM_SUPP_COUNTRY_LOC table of location records (For organization hierarchy level above the store/warehouse).

MODIFY

- Populate ITEM_SUPP_COUNTRY_LOC table of location records (For organization hierarchy level above the store/warehouse).

DELETE

- Populate ITEM_SUPP_COUNTRY_LOC table of location records (For organization hierarchy level above the store/warehouse).

Bulk or Single DML Module

All insert, update and delete SQL statements are located in the family packages. The private functions call these packages.

Filename: rmssub_item_sql

```
RMSSUB_ITEM_SQL.PERSIST_MESSAGE
               (O_error_message       IN OUT         VARCHAR2,
                I_item_rec            IN             RMSSUB_ITEM.ITEM_API_REC,
                I_message             IN             RIB_XItemDesc,
                I_message_type        IN             VARCHAR2)
```

This overloaded function checks the message type to route the object to the appropriate internal functions that perform DML insert and update processes.

ITEM CREATE

- Inserts a record in the ITEM_MASTER table
- Calls all the “create” functions to insert records into the following tables:
  - ITEM_SUPPLIER
  - ITEM_SUPP_COUNTRY
  - ITEM_SUPP_COUNTRY_LOC (optional)
  - PACKITEM (optional)
  - PACKITEM_BREAKOUT (optional)
  - VAT_ITEM (optional)
- Calls function to create initial item pricing information and provide for the insert into the PRICE_HIST table.

ITEM MODIFY

- Updates a record in the ITEM_MASTER table.

ITEM DELETE

- Inserts a record in the DAILY_PURGE table.
- Updates the status field for the record in the ITEM_MASTER table.

ITEM_SUPPLIER CREATE

- Inserts records in the ITEM_SUPPLIER table.

ITEM_SUPPLIER MODIFY

- Updates records in the ITEM_SUPPLIER table.

ITEM_SUPPLIER DELETE

- Deletes records from the ITEM_SUPPLIER table for item and if the delete children indicator in the message is Yes, deletes the records for the child items first.

ITEM_SUPP_COUNTRY CREATE

- Inserts records in the ITEM_SUPP_COUNTRY table.

ITEM_SUPP_COUNTRY MODIFY

- Updates records in the ITEM_SUPP_COUNTRY table.

ITEM_SUPP_COUNTRY DELETE

- Deletes records from the ITEM_SUPP_COUNTRY table for item and if the delete children indicator in the message is Yes, deletes the records for the child items first.

ITEM_SUPP_COUNTRY_LOC CREATE

- Bulk inserts records in the ITEM_SUPP_COUNTRY_LOC table for all locations indicated by the organization hierarchy values.

ITEM_SUPP_COUNTRY_LOC MODIFY

- Bulk updates records in the ITEM_SUPP_COUNTRY_LOC table for all locations indicated by the organization hierarchy values.

ITEM_SUPP_COUNTRY_LOC DELETE

- Bulk deletes records from the ITEM_SUPP_COUNTRY_LOC table for all locations indicated by the organization hierarchy values.

PACKITEM CREATE

- Inserts records in the PACKITEM table.
- Inserts records in the PACKITEM_BREAKOUT table.
- Updates ITEM_SUPP_COUNTRY_LOC and/or ITEM_SUPP_COUNTRY with calculated unit_cost.

VAT_ITEM CREATE

- Inserts records in the VAT_ITEM table and replaces any default records that were created from department/VAT.

VAT_ITEM DELETE

- Deletes records from the VAT_ITEM table.

Message DTD

Here are the filenames that correspond with each message type. Please consult the RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
XItemCre	Item Create Message	XItemDesc.dtd
XItemMod	Item Modify Message	XItemDesc.dtd
XItemDel	Item Delete Message	XItemRef.dtd
XItemSupCre	Item/Supplier Create Message	XItemDesc.dtd
XItemSupMod	Item/Supplier Modify Message	XItemDesc.dtd
XItemSupDel	Item/Supplier Delete Message	XItemRef.dtd
XItemSupCtyCre	Item/Supplier/Country Create Message	XItemDesc.dtd
XItemSupCtyMod	Item/Supplier/Country Modify Message	XItemDesc.dtd
XItemSupCtyDel	Item/Supplier/Country Delete Message	XItemRef.dtd
XISCLocCre	Item/Supplier/Country/Location Create Message	XItemDesc.dtd

Message Types	Message Type Description	Document Type Definition (DTD)
XISCLocMod	Item/Supplier/Country/Location Modify Message	XItemDesc.dtd
XISCLocDel	Item/Supplier/Country/Location Delete Message	XItemRef.dtd
XItemVatCre	Item/Vat Create Message	XItemDesc.dtd
XItemVatDel	Item/Vat Delete Message	XItemRef.dtd

Design Assumptions

Required fields are shown in RIB documentation.

Tables

RPM is called to set the initial pricing for the item. This populates tables in the RPM system.

TABLE	SELECT	INSERT	UPDATE	DELETE
ITEM_MASTER	Yes	Yes	Yes	No
ITEM_SUPPLIER	Yes	Yes	Yes	Yes
ITEM_SUPP_COUNTRY	Yes	Yes	Yes	Yes
ITEM_SUPP_COUNTRY_LOC	Yes	Yes	Yes	Yes
PRICE_HIST	No	Yes	No	No
PACKITEM	No	Yes	No	No
PACKITEM_BREAKOUT	No	Yes	No	No
VAT_ITEM	Yes	Yes	Yes	Yes
DAILY_PURGE	No	Yes	No	No
SYSTEM_OPTIONS	Yes	No	No	No
CHAIN	Yes	No	No	No
AREA	Yes	No	No	No
REGION	Yes	No	No	No
DISTRICT	Yes	No	No	No
STORE	Yes	No	No	No
WAREHOUSE	Yes	No	No	No
SUPS	Yes	No	No	No

Item Location

Functional Area

Items - Locations

Design Overview

The system option 'sor_item_ind' indicates whether RMS is the system of record for item maintenance. If RMS is the system of record, then item and item-location data is maintained in RMS. If RMS is not the system of record, then this API is used to import item-location data into RMS. This data can then be viewed in RMS, but not maintained.

Item locations can be maintained at the following levels of the organization hierarchy: chain, area, region, district, and store. Records are maintained for all stores within the location group. Because warehouses are not part of the organization hierarchy, they are only impacted by records maintained at the warehouse level. If building item-locations by organizational hierarchy, only locations in the hierarchy that exist on item-zone-price and do not already exist on item-location will be built.

Item locations can only be created for a single item. However, levels of the organization hierarchy are available for maintenance in order to facilitate location-level processing into RMS. The detail node is required for both create and modify messages.

Item supplier country locations will be created for the passed-in primary supplier/country if they do not already exist. If primary supplier/country locations are not passed in, then they will default from the item's primary supplier/country and a location will be created, if it does not already exist.

Item locations are required to be interfaced into RMS in active status. There is no delete function in this API. Instead, item locations can be put into inactive, discontinued, or deleted status. However, they will be deleted if the associated item is purged. If building item-locations by store or warehouse, then each passed-in location must exist on the item-zone-price and not already exist as an item-location.

Consume Module

Filename: rmssub_XITEMLOCs/b.pls

```
RMSSUB_XITEMLOC.CONSUME
              (O_status_code        IN OUT        VARCHAR2,
               O_error_message      IN OUT        RTK_ERRORS.RTK_TEXT%TYPE,
               I_message            IN            RIB_OBJECT,
               I_message_type       IN            VARCHAR2)
```

This procedure needs to initially ensure that the passed in message type is a valid type for item location messages. If the message type is invalid, a status of "E" should be returned to the external system along with an appropriate error message informing the external system that the status is invalid.

If the message type is valid, the generic RIB_OBJECT needs to be downcast to the actual object using the Oracle's treat function. If the downcast fails, a status of "E" should be returned to the external system along with an appropriate error message informing the external system that the object passed in is invalid.

If the downcast is successful, then consume needs to verify that the message passes all of the RMS business validation. It calls the RMSSUB_XITEMLOC_VALIDATE.CHECK_MESSAGE function to determine whether the message is valid. If the message passed RMS business validation, then the function will return true; otherwise, it will return false. If the message has failed RMS business validation, a status of "E" should be returned to the external system along with the error message returned from the CHECK_MESSAGE function.

Once the message has passed RMS business validation, it can be persisted to the RMS database. It calls the RMSSUB_XITEMLOC_SQL.PERSIST_MESSAGE() function. If the database persistence fails, the function returns false. A status of "E" should be returned to the external system along with the error message returned from the PERSIST_MESSAGE() function.

Once the message has been successfully persisted, there is nothing more for the consume procedure to do. A success status, "S", should be returned to the external system indicating that the message has been successfully received and persisted to the RMS database.

RMSSUB_XITEMLOC.HANDLE_ERROR() – This is the standard error handling function that wraps the API_LIBRARY.HANDLE_ERROR function.

Business Validation Module

It should be noted that some of the business validation is referential or involves uniqueness. This validation is handled automatically by the referential integrity constraints and the unique indexes implemented on the database.

Filename: rmssub_xitemlocvals/b.pls

```
RMSSUB_XITEMLOC_VALIDATE.CHECK_MESSAGE
               (O_error_message       IN OUT         VARCHAR2,
                O_ITEMLOC_rec         OUT            ITEMLOC_REC,
                I_message             IN             RIB_XItemLocDesc,
                I_message_type        IN             VARCHAR2)
```

This function performs all business validation associated with message and builds the item locations record for persistence.

ITEMLOC CREATE

- Check required fields
- Verify primary supplier/country exists on Item-supplier-country
- If creating locations by store or warehouse, verify each value has an item-zone-price record.
- If creating locations by store or warehouse, verify passed in locations do not currently exist.
- If item is a buyer pack, verify receive as type is valid based on item's order as type.
- Default required fields not provided (store order multiple, taxable indicator, local item description, primary supplier/country, receive as type).
- Build item-location records.
- Build price history records.

ITEMLOC MODIFY

- Check required fields
- Populate item-location record.

Bulk or single DML module

Filename: rmssub_xitemlocsqls/b.pls

```
RMSSUB_XITEMLOC_SQL.PERSIST
               (O_error_message       IN OUT         VARCHAR2,
                I_dml_rec             IN             ITEMLOC_RECTYPE ,
                I_message             IN             RIB_XITEMLOCDesc)
```

ITEMLOC CREATE

- Insert a record into the item-location table.
- Insert a record into the item-location-stock on hand table
- If necessary, insert a record into the item supplier country location table.
- Insert a record into the price history table.

ITEMLOC MODIFY

- Update item-location table.

Message DTD

Here are the filenames that correspond with each message type. Please consult the RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Type	Message Type Description	DTD Name
xitemloccre	External item locations create	XItemlocDesc.dtd
xitemlocMod	External item locations odification	XItemlocDesc.dtd

Tables

TABLE	SELECT	INSERT	UPDATE	DELETE
ITEM_SUPP_COUNTRY	Yes	No	No	No
ITEM_SUPP_COUNTRY_LOC	Yes	Yes	No	No
ITEM_LOC_SOH	Yes	Yes	Yes	No
STORE	Yes	No	No	No
WH	Yes	No	No	No
ITEM_LOC	Yes	Yes	Yes	No
SYSTEM_OPTIONS	Yes	No	No	No
PRICE_HIST	No	Yes	No	No
ITEM_MASTER	Yes	No	No	No
POS_MODS	No	Yes	No	No
PACKITEM_BREAKOUT	Yes	No	No	No
CHAIN	Yes	No	No	No
AREA	Yes	No	No	No

TABLE	SELECT	INSERT	UPDATE	DELETE
REGION	Yes	No	No	No
DISTRICT	Yes	No	No	No
PACKITEM	Yes	No	No	No
RPM_ITEM_ZONE_PRICE	Yes	No	No	No
PRICE_ZONE_GROUP_STORE	Yes	No	No	No
CURRENCIES	Yes	No	No	No
ELC_TABLES	Yes	No	No	No
VAT_ITEM	Yes	No	No	No
PARTNER	Yes	No	No	No

Item Reclassification

Functional Area

Items - Reclassification

Design Overview

RMS subscribes to item reclassification messages that are published by an external system. This subscription is necessary in order to keep RMS in sync with the external system. The retailer can view the pending reclassifications online in RMS.

This API allows external systems to create and delete item reclassification events within RMS.

At least one detail must be passed for a valid reclassification message. Reclassification items can be created or deleted within the reclassification message. Reclass item creates will send a snapshot of the reclass event. However, reclass item deletes do not require any header information as items are unique for reclassification and items may be deleted across reclass events.

Only level one items can be interfaced via this API. If the item is a pack, only non-simple packs can be interfaced. Simple pack items will be reclassified when their component is reclassified.

During the reclassification batch process, it will determine if any pack items exist in RMS that contain the items or any of that item's children being reclassified. If such a pack exists and contains no other items, the batch process adds the pack to the reclassification event being created in RMS.

It is valid for a reclassification event to be created for a department/class/subclass not yet existing but planning to exist. This is valid as long as they department/class/subclass is scheduled to be created on or prior to the reclassification taking effect.

Deleting reclassifications can either occur by:

- Items on a reclass event or across events.
- A single reclassification event.
- All reclassification events on a particular event date (deletion through the use of the reclass_date may result in the deletion of numerous reclass events).
- All reclassification events.

Deleting a reclassification header will require either a reclass no, reclass date, or purge all ind.

Consume Module

Filename: rmssub_XITEMRCLSs/b.pls

```
RMSSUB_XITEMRCLS.CONSUME
               (O_status_code        IN OUT        VARCHAR2,
                O_error_message      IN OUT        RTK_ERRORS.RTK_TEXT%TYPE,
                I_message            IN            RIB_OBJECT,
                I_message_type       IN            VARCHAR2)
```

This procedure needs to initially ensure that the passed in message type is a valid type for item reclassification messages. If the message type is invalid, a status of "E" should be returned to the external system along with an appropriate error message informing the external system that the status is invalid.

If the message type is valid, the generic RIB_OBJECT needs to be downcast to the actual object using the Oracle's treat function. If the downcast fails, a status of "E" should be returned to the external system along with an appropriate error message informing the external system that the object passed in is invalid.

If the downcast is successful, then consume needs to verify that the message passes all of the RMS business validation. It calls the RMSSUB_XITEMRCLS_VALIDATE.CHECK_MESSAGE function to determine whether the message is valid. If the message passed RMS business validation, then the function will return true; otherwise, it will return false. If the message has failed RMS business validation, a status of "E" should be returned to the external system along with the error message returned from the CHECK_MESSAGE function.

Once the message has passed RMS business validation, it can be persisted to the RMS database. It calls the RMSSUB_XITEMRCLS_SQL.PERSIST_MESSAGE() function. If the database persistence fails, the function returns false. A status of "E" should be returned to the external system along with the error message returned from the PERSIST_MESSAGE() function.

Once the message has been successfully persisted, there is nothing more for the consume procedure to do. A success status, "S", should be returned to the external system indicating that the message has been successfully received and persisted to the RMS database.

RMSSUB_XITEMRCLS.HANDLE_ERROR() – This is the standard error handling function that wraps the API_LIBRARY.HANDLE_ERROR function.

Business Validation Module

It should be noted that some of the business validation is referential or involves uniqueness. This validation is handled automatically by the referential integrity constraints and the unique indexes implemented on the database.

Filename: rmssub_xitemrclsvals/b.pls

```
RMSSUB_XITEMRCLS_VALIDATE.CHECK_MESSAGE
               (O_error_message      IN OUT        VARCHAR2,
                O_ITEMRCLS_rec       OUT           ITEMRCLS_REC,
                I_message            IN            RIB_XITEMRCLSDesc,
                I_message_type       IN            VARCHAR2)
```

This function performs all business validation associated with message and builds the item reclassification record for persistence.

ITEMRCLS CREATE

- Check required fields.
- Verify items not on existing reclassification.
- Validate the reclassification date (must be today or greater).
- Verify hierarchy of item being reclassified to (either an existing hierarchy or a pending hierarchy that will be created prior to the item reclassification).
- Verify non-consignment related reclassification and no unit and dollar stocks performed on items.
- Build reclassification records.

ITEMRCLS DELETE

- Check required fields.
- For reclassification header deletes, verify deleting by either reclassification number, reclassification (event) date, or purging all reclassifications.
- Populate record.

ITEMRCLS DETAIL CREATE

- Check required fields.
- Verify items not on existing reclassification.
- Validate the reclassification date (must be today or greater).
- Verify hierarchy of item being reclassified to (either an existing hierarchy or a pending hierarchy that will be created prior to the item reclassification).
- Verify non-consignment related reclassification and no unit and dollar stocks performed on items.
- Build reclassification records.

ITEMRCLS DETAIL DELETE

- Check required fields.
- Populate record.

Bulk or Single DML Module

Filename: rmssub_XITEMRCLSsqls/b.pls

```
RMSSUB_XITEMRCLS_SQL.PERSIST
               (O_error_message       IN OUT         VARCHAR2,
                I_dml_rec             IN             ITEMRCLS_RECTYPE ,
                I_message             IN             RIB_XITEMRCLSDesc)
```

ITEMRCLS CREATE

- Insert a record into the reclass header table.
- Insert a record into the reclass item table.

ITEMRCLS DETAIL DELETE

- Delete from the reclass item table.

ITEMRCLS DELETE

- If purging all records, delete all from reclass item table.
- If purging all records, delete all from reclass header table.
- If not purging, delete from reclass item for reclass number or all reclass for an event date.
- If not purging, delete from reclass header for reclass number or all reclass for an event date.

ITEMRCLS DELETE

- Delete from reclass item for all items on record.
- If no items exist for an event, delete the reclass event.

Message DTD

Here are the filenames that correspond with each message type. Please consult the RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Type	Message Type Description	Document Type Definition (DTD)
xitemrclscre	External item reclassification create	XItemRclsDesc.dtd
xitemrclsdtlcre	External item reclassification detail create	XItemRclsDesc.dtd
Xitemrclsdel	External item reclassification delete	XitemRclsRef.dtd
Xitemrclsdtldel	External item reclassification detail delete	XItemRclsRef.dtd

Design Assumptions

Orderable buyer packs as ‘E’aches will not be allowed to be reclassified if department level ordering is Y in RMS.

Tables

TABLE	SELECT	INSERT	UPDATE	DELETE
RECLASS_HEAD	Yes	Yes	No	Yes
RECLASS_ITEM	Yes	Yes	No	Yes
ITEM_MASTER	Yes	No	No	No
PACKITEM	Yes	No	No	No
PACKITEM_BREAKOUT	Yes	No	No	No
V_MERCH_HIER	Yes	No	No	No

Location Trait

Functional Area

Location Trait

Design Overview

The RMS system option 'sor_org_hier_ind' indicates whether RMS is the system of record for organization hierarchy maintenance, including location traits. If RMS is the system of record, then RMS databases hold that information and it is entered and maintained using the RMS windows. If RMS is not the system of record, then that information is imported into RMS from an external system, and the information can be viewed but not maintained on the RMS windows.

When RMS is not the system of record for location traits, the Location Trait Subscription API processes incoming data from an external system to create, edit and delete location traits in RMS. This data is processed immediately upon message receipt so success or failure can be communicated to the external application. For a general discussion of the RMS general hierarchy, see the chapter "Organization Hierarchy Batch" in volume 1 of this RMS Operations Guide.

Consume Module

Filename: rmssub_xloctrts/b.pls

```
RMSSUB_XLOCTRT.CONSUME
               (O_status_code        IN OUT        VARCHAR2,
                O_error_message      IN OUT        VARCHAR2,
                I_message            IN            RIB_OBJECT,
                I_message_type       IN            VARCHAR2)
```

This procedure will need to initially ensure that the passed in message type is a valid type for loc traits messages. If the message type is invalid, a status of "E" should be returned to the external system along with an appropriate error message informing the external system that the status is invalid.

If the message type is valid, the generic RIB_OBJECT need to be downcast to the actual object using the Oracle's treat function. If the downcast fails, a status of "E" should be returned to the external system along with an appropriate error message informing the external system that the object passed in is invalid.

If the downcast is successful, then consume needs to verify that the message passes all of the RMS business validation. It calls the RMSSUB_XLOCTRT_VALIDATE.CHECK_MESSAGE function to determine whether the message is valid. If the message passed RMS business validation, then the function will return true, otherwise it will return false. If the message has failed RMS business validation, a status of "E" should be returned to the external system along with the error message returned from the CHECK_MESSAGE function.

Once the message has passed RMS business validation, it can be persisted to the RMS database. It calls the RMSSUB_XLOCTRT_SQL.PERSIST_MESSAGE() function. If the database persistence fails, the function will return false. A status of "E" should be returned to the external system along with the error message returned from the PERSIST_MESSAGE() function.

Once the message has been successfully persisted, there is nothing more for the consume procedure to do. A success, “S”, status should be returned to the external system indicating that the message has been successfully received and persisted to the RMS database.

RMSSUB_XLOCTRT.HANDLE_ERROR() – This is the standard error handling function that wraps the API_LIBRARY.HANDLE_ERROR function.

Business Validation Module

It should be noted that some of the business validation is referential or involves uniqueness. This validation is handled automatically by the referential integrity constraints and the unique indexes implemented on the database.

Filename: rmssub_xloctrtvals/b.pls

```
RMSSUB_XLOCTRT_VALIDATE.CHECK_MESSAGE
               (O_error_message       IN OUT         VARCHAR2,
                O_loctrait_rec        OUT            LOC_TRAITS_REC,
                I_message             IN             RIB_XLocTraitDesc,
                I_message_type        IN             VARCHAR2)
```

This function performs all business validation associated with messages and builds the location trait record for persistence.

LOCATION TRAIT CREATE

- Check required fields.
- Populate record with message data.

LOCATION TRAIT MODIFY

- Check required fields.
- Verify the location trait exists.
- Populate record with message data.

LOCATION TRAIT DELETE

- Check required fields.
- Verify the location trait exists.
- Populate record with message data.

Bulk or single DML module

All insert, update and delete SQL statements are located in the family package. This package is LOC_TRAITS_SQL. The private functions will call this package.

Filename: rmssub_xloctrtsqls/b.pls

```
RMSSUB_XLOCTRT_SQL.PERSIST_MESSAGE
                (O_error_message      IN OUT         VARCHAR2,
                I_loc_trait_rec       IN             LOC_TRAIT_REC,
                I_message_type        IN             VARCHAR2,)
```

This function determines what type of database transaction it will call based on the message type.

LOCATION TRAIT CREATE

- Create messages get added to the location trait table.

LOCATION TRAIT MODIFY

- Modify messages directly update the location trait table with changes.

LOCATION TRAIT DELETE

- Delete messages directly remove location trait records.

Message DTD

Here are the filenames that correspond with each message type. Please consult RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Type	Message Type Description	DTD Name
xloctrtcre	External Location Trait Create	XLocTrtDesc.dtd
xloctrtdel	External Location Trait Delete	XLocTrtRef.dtd
xloctrtmod	External Location Trait Modification	XLocTrtDesc.dtd

Design Assumptions

Required fields are shown in RIB documentation.

Tables

TABLE	SELECT	INSERT	UPDATE	DELETE
LOC_TRAITS	Yes	Yes	Yes	Yes

Merchandise Hierarchy

Functional Area

Merchandise Hierarchy

- Company
- Division
- Group
- Department
- Class
- Subclass

Design Overview

The merchandise hierarchy allows the retailer to create the relationships that are necessary to support the product management structure of a company. This hierarchy reflects a classification of merchandise into multi-level descriptive categorizations to facilitate the planning, tracking, reporting, and management of merchandise within the company.

The levels of the structure are truly hierarchical, meaning that at any given level, the applicable item can belong to one and only one categorization.

System of Record

The sor_merch_hier_ind system option indicates whether RMS is the system of record for merchandise hierarchy maintenance. If RMS is the system of record, then RMS databases hold that information. If RMS is not the system of record, then that information is imported into RMS from outside systems. When the value of the indicator is N (No), retailers can no longer create or delete values from the hierarchy online in RMS. They can only view the values.

When RMS is not the system of record, it may subscribe to the merchandise hierarchy subscription API. The subscription keeps RMS in sync with the external system that is responsible for maintaining the merchandise hierarchy.

RMS will expose an API that will allow external systems to create and edit a company: create, edit, and delete a division, group, department, class, and subclass.

A company cannot be deleted, and only one company can exist in RMS. Creates and edits of all levels of the merchandise hierarchy occur immediately upon receipt of the message. Division and group deletes also occur immediately upon receipt of the message.

Departments, classes, and subclasses will not actually be deleted from the system upon receipt of the message. Instead, they will be added to the DAILY_PURGE table. However, validation will occur to ensure the records can be deleted. When the daily purge batch process runs, the records will be removed from the system. Because these levels are not deleted when the message is received, the API will not be able to communicate whether the removal of the record from RMS has been successful, only that the message was successfully received.

Department VAT records can be created and edited within the department message (VAT records are not deleted). VAT creates can be passed in with a department create message, or they can be passed in with their own specific message type. The VAT create and modify messages will send a snapshot of the department record. Note that VAT region and VAT codes records must exist prior to creating department VAT records. Also, when passing in a new VAT region to an existing department with attached items, the VAT will default to all items.

No other levels of the hierarchy have detail nodes associated with them.

The merchandise hierarchy must be created from the highest level down. Conversely, the hierarchy must be deleted from the lowest level up. Each lower level references a parent level (except for group/company). This means a department is associated with a group; a class is associated with a department; and a subclass is associated with department/class combination (because classes are not unique across departments while departments are unique independent of groups).

Consume Module

Rmssub_xmrchhrs/b.pls

```
RMSSUB_XMRCHHR.CONSUME
               (O_status_code        IN OUT        VARCHAR2,
                O_error_message      IN OUT        VARCHAR2,
                I_message            IN            RIB_OBJECT,
                I_message_type       IN            VARCHAR2)
```

This procedure will call the appropriate merchandise hierarchy family package based on the message type passed in.

Any company message type will call RMSSUB_XMRCHHRCOMP.CONSUME.

Any division message type will call RMSSUB_XMRCHHRDIV.CONSUME.

Any group message type will call RMSSUB_XMRCHHRGRP.CONSUME.

Any department message type will call RMSSUB_XMRCHHRDEPT.CONSUME.

Any class message type will call RMSSUB_XMRCHHRSCLS.CONSUME.

Any subclass message type will call RMSSUB_XMRCHHRCLS.CONSUME.

Each family consume does the following steps.

They initially ensure that the passed in message type is a valid type for family messages. If the message type is invalid, a status of ‘E’ should be returned to the external system along with an appropriate error message informing the external system that the status is invalid.

If the message type is valid, the generic RIB_OBJECT will need to be downcast to the actual object using the Oracle’s treat function. If the downcast fails, a status of ‘E’ should be returned to the external system along with an appropriate error message informing the external system that the object passed in is invalid.

If the downcast is successful, then consume will need to verify that the message passes all of the RMS business validation. If the message has failed RMS business validation, a status of ‘E’ should be returned to the external system along with the error message returned from the CHECK_MESSAGE function.

Once the message has passed RMS business validation, it can be persisted to the RMS database. If the database persistence fails, the function will return false. A status of ‘E’ should be returned to the external system along with the error message returned from the PERSIST_MESSAGE() function.

Once the message has been successfully persisted, there is nothing more for the consume procedure to do. A success status, 'S', should be returned to the external system indicating that the message has been successfully received and persisted to the RMS database.

RMSSUB_XMRCHHR.HANDLE_ERROR() – This is the standard error handling function that wraps the API_LIBRARY.HANDLE_ERROR function.

Business Validation Module

It should be noted that some of the business validation is referential or involves uniqueness. This validation is handled automatically by the referential integrity constraints and the unique indexes implemented on the database.

Filename: rmssub_xmrchhr[family_name]vals/b.pls

```
RMSSUB_XMRCHHR[family_name]_VALIDATE.CHECK_MESSAGE
       (O_error_message      IN OUT         VARCHAR2,
        O_[family_name]_rec  OUT            NOCOPY  MERCH_SQL.[FAMILY_NAME]_TYPE,
        I_message            IN             RIB_XMrchHr[family_name]Desc,
        I_message_type       IN             VARCHAR2)
```

This function performs all business validation associated with messages and builds the merchandise hierarchy record for persistence.

COMPANY CREATE

- Check required fields.
- Verify a company record does not already exist.
- Populate record with message data.

COMPANY MODIFY

- Check required fields.
- Verify the company exists.
- Populate record with message data.

DIVISION CREATE

- Check required fields.
- Verify total market amount.
- Populate record with message data.

DIVISION MODIFY

- Check required fields.
- Verify division record exists.
- Verify total market amount.
- Populate record with message data.

DIVISION DELETE

- Check required fields.
- Verify division record exists.
- Populate record with message data.

GROUP CREATE

- Check required fields.
- Populate record with message data.

GROUP MODIFY

- Check required fields.
- Verify division record exists.
- Populate record with message data.

GROUP DELETE

- Check required fields.
- Verify division record exists.
- Populate record with message data.

DEPARTMENT CREATE

- Check required fields. A department cannot be set up as both direct cost and consignment. Either the budget markup percent or the budget intake percent must be passed in. If RPM is installed, the average tolerance percent and maximum average counter must be greater than zero.
- Verify total market amount.
- Check required child nodes.
- Populate record with message data.
- Populate default fields.

DEPARTMENT MODIFY

- Check required fields.
- Verify total market amount.
- Verify department record exists
- Populate record with message data.
- Populate default fields.

DEPARTMENT DELETE

- Check required fields.
- Verify division record exists.
- Verify a department can be deleted. Note that this check performs the same validation that occurs before a department is actually deleted in the batch process.
- Populate record with message data.

CLASS CREATE

- Check required fields.
- Populate record with message data.
- Populate default fields.

CLASS MODIFY

- Check required fields.
- Verify class record exists.
- Populate record with message data.

CLASS DELETE

- Check required fields.
- Verify class record exists.
- Verify a class can be deleted. Note that this check performs the same validation that occurs before a class is actually deleted in the batch process.
- Populate record with message data.

SUBCLASS CREATE

- Check required fields.
- Populate record with message data.
- Populate default fields.

SUBCLASS MODIFY

- Check required fields.
- Verify subclass record exists.
- Populate record with message data.

SUBCLASS DELETE

- Check required fields.
- Verify subclass record exists.
- Verify a subclass can be deleted. Note that this check performs the same validation that occurs before a subclass is actually deleted in the batch process.
- Populate record with message data.

Bulk or Single DML Module

All insert, update and delete SQL statements are located in the family package. This package is MERCH_SQL. The private functions will call this package.

Filename: rmssub_xmrchhr[family_name]sqls/b.pls

```
RMSSUB_XMRCHHR[family_name]__SQL.PERSIST_MESSAGE
               (O_error_message       IN OUT         VARCHAR2,
                I_[family_name]_rec   IN MERCH_SQL.[FAMILY_NAME]_TYPE,
                I_message_type        IN             VARCHAR2,)
```

This function determines what type of database transaction it will call based on the message type.

COMPANY CREATE

- Create messages get added to the company table.

COMPANY MODIFY

- Modify messages directly update the company table with changes.

DIVISION CREATE

- Create messages get added to the division table.

DIVISION MODIFY

- Modify messages directly update the division table with changes.

DIVISION DELETE

- Delete messages directly remove division records.

GROUP CREATE

- Create messages get added to the group table.

GROUP MODIFY

- Modify messages directly update the group table with changes.

GROUP DELETE

- Delete messages directly remove group records.

DEPARTMENT CREATE

- Create messages get added to the department table. VAT details may also be added for a create message.

DEPARTMENT MODIFY

- Modify messages directly update the department table with changes.

VAT CREATE

- Add VAT records to department VAT table.

VAT MODIFY

- Update VAT records on department VAT table.

DEPARTMENT DELETE

- Department gets added to purging table to be processed in batch cycle.

CLASS CREATE

- Create messages get added to the class table.

CLASS MODIFY

- Modify messages directly update the class table with changes.

CLASS DELETE

- Class gets added to purging table to be processed in batch cycle

SUBCLASS CREATE

- Create messages get added to the subclass table.

SUBCLASS MODIFY

- Modify messages directly update the subclass table with changes.

SUBCLASS DELETE

- Subclass gets added to purging table to be processed in batch cycle.

Message DTD

Here are the filenames that correspond with each message type. Please consult the RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Type	Message Type Description	DTD Name
xmrchhrclscre	External Create Class	XMrchHrClsDesc.dtd
xmrchhrcompcre	External Create Company	XMrchHrCompDesc.dtd
xmrchhrdeptcre	External Create Department	XMrchHrDeptDesc.dtd
xmrchhrdivcre	External Create Division	XMrchHrDivDesc.dtd
xmrchhrgrpcre	External Create Group	XMrchHrGrpDesc.dtd
xmrchhrsclscre	External Create Subclass	XMrchHrSclsDesc.dtd
xmrchhrclsdel	External Delete Class	XMrchHrClsRef.dtd
xmrchhrdeptdel	External Delete Department	XMrchHrDeptRef.dtd
xmrchhrdivdel	External Delete Division	XMrchHrDivRef.dtd
xmrchhrgrpdel	External Delete Group	XMrchHrGrpRef.dtd
xmrchhrsclsdel	External Delete Subclass	XMrchHrSclsRef.dtd
xmrchhrvatcre	External Merch Hierarchy VAT create	XMrchHrDeptDesc.dtd
xmrchhrvatmod	External Merch Hierarchy VAT modify	XMrchHrDeptDesc.dtd
xmrchhrclsmod	External Modify Class	XMrchHrClsDesc.dtd
xmrchhrcompmod	External Modify Company	XMrchHrCompDesc.dtd
xmrchhrdeptmod	External Modify Department	XMrchHrDeptDesc.dtd
xmrchhrdivmod	External Modify Division	XMrchHrDivDesc.dtd
xmrchhrgrpmod	External Modify Group	XMrchHrGrpDesc.dtd
xmrchhrsclsmod	External Modify Subclass	XMrchHrSclsDesc.dtd

Performance / Volume Considerations

The API should be capable of processing 120 messages per hour.

Design Assumptions

Required fields are shown in RIB documentation.

Tables

Note that this section does **not** include the tables checked in VALIDATE_RECORDS_SQL.DEL_DEPS/CLASS/SUBCLASS.

TABLE	SELECT	INSERT	UPDATE	DELETE
COMPHEAD	Yes	Yes	Yes	No
DIVISION	Yes	Yes	Yes	Yes
DAILY_PURGE	No	Yes	No	No
GROUPS	Yes	Yes	Yes	Yes
DEPS	Yes	Yes	Yes	No
VAT_DEPS	Yes	Yes	Yes	No
CLASS	Yes	Yes	Yes	No
SUBCLASS	Yes	Yes	Yes	No

Merchandise Hierarchy Reclassification

Functional Area

Merchandise Hierarchy Reclassification

Design Overview

RMS can subscribe to merchandise hierarchy reclassification messages that are published by an external system. This subscription is necessary in order to keep RMS in sync with the external system.

Retailers can view pending reclassifications online. Users with the appropriate security level can edit pending reclassifications and their effective dates. For information about batch processing in this functional area, see the chapter “Reclassification Batch” in volume 1 of this RMS Operations Guide.

This API will subscribe to future effective dated merchandise hierarchy additions and changes through the merchandise hierarchy reclassification create, update and delete messages.

In addition to the existing RMS merchandise hierarchy tables which hold the current merchandise hierarchy information in RMS, a pending merchandise hierarchy table will hold the future merchandise hierarchy creation or modification (but not deletion). These pending merchandise hierarchy reclassification events can be created, modified or deleted via this API. A separate batch process will read the information off the pending merchandise hierarchy table and create or modify the merchandise hierarchy information in RMS once the change effective date arrives. This batch process is **not** covered in this design document.

For a given merchandise hierarchy, all pending merchandise reclassification events have to be for the same effective date. A modify message can potentially change a pending reclassification’s effective date.

This API would **not** accept messages to delete an existing merchandise hierarchy. Any deletion should be done through the merchandise hierarchy subscription API instead. Furthermore, this API would **not** allow moving a class or subclass between departments. By allowing it, the system would essentially be reclassifying items, and there is a separate item reclassification process for it.

Consume Module

Rmssub_xmrchhrclss/b.pls

```
RMSSUB_XMRCHHRRCLS.CONSUME
               (O_status_code        IN OUT        VARCHAR2,
                O_error_message      IN OUT        VARCHAR2,
                I_message            IN            RIB_OBJECT,
                I_message_type       IN            VARCHAR2)
```

This procedure will need to initially ensure that the passed in message type is a valid type for merchandise hierarchy reclassification messages. If the message type is invalid, a status of 'E' should be returned to the external system along with an appropriate error message informing the external system that the status is invalid.

If the message type is valid, the generic RIB_OBJECT will need to be downcast to the actual object using the Oracle's treat function. If the downcast fails, a status of 'E' should be returned to the external system along with an appropriate error message informing the external system that the object passed in is invalid.

If the downcast is successful, then consume will need to verify that the message passes all of the RMS business validation. If the message has failed RMS business validation, a status of 'E' should be returned to the external system along with the error message returned from the CHECK_MESSAGE function.

Once the message has passed RMS business validation, it can be persisted to the RMS database. If the database persistence fails, the function will return false. A status of 'E' should be returned to the external system along with the error message returned from the PERSIST_MESSAGE() function.

Once the message has been successfully persisted, there is nothing more for the consume procedure to do. A success status, 'S', should be returned to the external system indicating that the message has been successfully received and persisted to the RMS database.

RMSSUB_ XMRCHHRRCLS.HANDLE_ERROR() – This is the standard error handling function that wraps the API_LIBRARY.HANDLE_ERROR function.

Business Validation Module

It should be noted that some of the business validation is referential or involves uniqueness. This validation is handled automatically by the referential integrity constraints and the unique indexes implemented on the database.

Filename: rmssub_xmrchhrrclsvals/b.pls

```
RMSSUB_XMRCHHRRCLS_VALIDATE.CHECK_MESSAGE
       (O_error_message      IN OUT         VARCHAR2,
        O_pend_merch_hier_rec        OUT            PEND_MERCH_HIER%ROWTYPE,
            I_message                IN             RIB_XMrchHrRclsDesc_REC,
            I_message_type           IN             VARCHAR2)
```

This function performs all business validation associated with the messages and builds the merchandise hierarchy record for persistence.

CREATE

- Check required fields. Required fields vary based on hierarchy level.

  Adding New Hierarchy

  - Verify passed in hierarchy does not already exist.
  - Verify parent hierarchy already exists on merchandise hierarchy or pending merchandise hierarchy tables.

  Modifying Existing Hierarchy

  - Verify passed in hierarchy does already exist.
  - Verify that class and subclass hierarchies have passed in parent hierarchy in an existing hierarchy (i.e. classes and subclasses are not allowed to be reclassified into another department).
- Populate record with message data

MODIFY

- Check required fields.
- Verify the hierarchy is already pending.
- Populate record with message data.

DELETE

- Check required fields.
- Verify a pending hierarchy event exists.
- Verify no pending hierarchy events exist for levels below the passed in hierarchy level.
- Populate record with message data.

Bulk or Single DML Module

All insert, update and delete SQL statements are located in the family package. This package is MERCH_RECLASS_SQL. The private functions will call this package.

Filename: rmssub_ xmrchhrrclssqls /b.pls

```
RMSSUB_XMRCHHRRCLS_SQL.PERSIST_MESSAGE
          (O_error_message          IN OUT        VARCHAR2,
        I_pend_merch_hier_rec       IN            PEND_MERCH_HIER%ROWTYPE,
        I_message_type              IN            VARCHAR2)
```

This function determines what type of database transaction it will call based on the message type.

CREATE

- Create messages get added to the pending merchandise hierarchy table.

MODIFY

- Modify messages directly update the pending merchandise hierarchy table with changes.

DELETE

- Delete messages get removed from the pending merchandise hierarchy table.

Message DTD

Here are the filenames that correspond with each message type. Please consult RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Type	Message Type Description	DTD Name
xmrchhrrclscre	Create Merchandise Hierarchy Reclassification	XMrchHrRclsDesc.dtd
xmrchhrrclsdel	Delete Merchandise Hierarchy Reclassification	XMrchHrRclsRef.dtd
xmrchhrrclsmod	Modify Merchandise Hierarchy Reclassification	XMrchHrRclsDesc.dtd

Design Assumptions

Required fields are shown in RIB documentation.

The values for passing in the hierarchy and action type are outlined in the RIB documentation.

Tables

TABLE	SELECT	INSERT	UPDATE	DELETE
COMPHEAD	Yes	No	No	No
DIVISION	Yes	No	No	No
GROUPS	Yes	No	No	No
DEPS	Yes	No	No	No
CLASS	Yes	No	No	No
SUBCLASS	Yes	No	No	No
PEND_MERCH_HIER	Yes	Yes	Yes	Yes

Organizational Hierarchy

Functional Area

Organizational Hierarchy

- Chain
- Area
- Region
- District

Design Overview

The system of record hierarchy indicator (sor_org_hier_ind) system option indicates whether RMS is the system of record for organization hierarchy maintenance. If RMS is the system of record, then RMS databases hold that information. If RMS is not the system of record, then that information is imported into RMS from outside systems. When the value of the indicator is 'N' (No), users can no longer create or modify values from the hierarchy online in RMS. They can only view the values.

When RMS is not the system of record, it can take advantage of the organization hierarchy subscription API. The subscription keeps RMS in sync with the external system that is responsible for maintaining the organization hierarchy. Although stores are part of the organization hierarchy, they differ sufficiently to require their own subscription API.

The organization hierarchy subscription also assigns existing location traits to or deletes them from elements of the organization hierarchy.

RMS will expose an API that will allow external systems to create, edit, and delete chain, area, region, and districts. All creates, updates, and deletes will occur immediately upon receipt of the message.

Location trait records are created and deleted within the area, region, and district messages. Location trait creates can be passed in with the area, region, or district create message, or they can be passed in with their own specific create message type attached to the aforementioned messages. The location trait create messages will send a snapshot of the hierarchy record they are attached to. Location trait delete messages will be processed separate from the hierarchy delete messages.

Note that location trait records must exist prior to attaching them to any hierarchy.

Chains do not have location traits associated with them.

The organizational hierarchy must be created from the highest level down. Conversely, the hierarchy must be deleted from the lowest level up. Each lower level references a parent level. This means an area is associated with a chain, a region is associated with an area, and a district is associated with a region. Location traits must be removed from a hierarchy before it can be removed.

For a general discussion of the organization hierarchy in RMS, see the chapter "Organization Hierarchy Batch" in volume 1 of this RMS Operations Guide. For more information about location traits, see the sections 'Location Trait' and 'Store' in the chapter "Subscription Design" in this volume of the RMS Operations Guide.

Consume Module

rmssub_xorghiers/b.pls

```
RMSSUB_XORGHIER.CONSUME
               (O_status_code        IN OUT        VARCHAR2,
                O_error_message      IN OUT        VARCHAR2,
                I_message            IN            RIB_OBJECT,
                I_message_type       IN            VARCHAR2)
```

This procedure will need to initially ensure that the passed in message type is a valid type for organizational hierarchy messages. The valid message types for organizational hierarchy messages are listed in a section below.

If the message type is invalid, a status of “E” should be returned to the external system along with an appropriate error message informing the external system that the status is invalid.

If the message type is valid, the generic RIB_OBJECT needs to be downcast to the actual object using the Oracle’s treat function. There will be an object type that corresponds with each message type. If the downcast fails, a status of “E” is returned to the external system along with an appropriate error message informing the external system that the object passed in is invalid.

If the downcast is successful, then consume needs to verify that the message passes all of the RMS business validation. It calls the RMSSUB_XORGHIER_VALIDATE.CHECK_MESSAGE function to determine whether the message is valid. If the message passed RMS business validation, then the function returns true; otherwise, it returns false. If the message fails RMS business validation, a status of “E” is returned to the external system along with the error message returned from the CHECK_MESSAGE function.

Once the message has passed RMS business validation, it is persisted to the RMS database. It calls the RMSSUB_XORGHIER_SQL.PERSIST_MESSAGE() function. If the database persistence fails, the function returns false. A status of “E” is returned to the external system along with the error message returned from the PERSIST_MESSAGE() function.

Once the message has been successfully persisted, there is nothing more for the consume procedure to do. A success status, “S”, is returned to the external system indicating that the message has been successfully received and persisted to the RMS database.

RMSSUB_ XORGHIER.HANDLE_ERROR() – This is the standard error handling function that wraps the API_LIBRARY.HANDLE_ERROR function.

Business Validation Module

It should be noted that some of the business validation is referential or involves uniqueness. This validation is handled automatically by the referential integrity constraints and the unique indexes implemented on the database.

Filename rmssub_xohvals/b.pls

```
RMSSUB_XORGHIER_VALIDATE.CHECK_MESSAGE
       (O_error_message   IN OUT     VARCHAR2,
        O_org_hier_rec    OUT        NOCOPY  ORGANIZATION_SQL.ORG_HIER_REC,
        I_message         IN         RIB_XOrgHrDesc,
        I_message_type    IN         VARCHAR2)
```

This function performs all business validation associated with messages and builds the organizational hierarchy record for persistence.

CHAIN CREATE

- Check required fields.
- Check hierarchy level.
- Populate record with message data.

CHAIN MODIFY

- Check required fields.
- Verify the chain exists.
- Populate record with message data.

CHAIN DELETE

- Check required fields.
- Verify the chain exists.
- Check if chain is currently used on the regionality tables.
- Populate record with message data.

AREA CREATE

- Check required fields.
- Check hierarchy level.
- Populate location traits.
- Populate record with message data.

AREA MODIFY

- Check required fields.
- Check hierarchy level.
- Verify area record exists.
- Populate record with message data.

AREA DELETE

- Check required fields.
- Verify the area exists.
- Check if area is currently used on the regionality tables.
- Populate record with message data.

REGION CREATE

- Check required fields.
- Check hierarchy level.
- Populate location traits.
- Populate record with message data.

REGION MODIFY

- Check required fields.
- Check hierarchy level.
- Verify region record exists.
- Get the existing area for region.
- Populate record with message data.

REGION DELETE

- Check required fields.
- Verify the region exists.
- Check if region is currently used on the regionality tables.
- Populate record with message data.

DISTRICT CREATE

- Check required fields.
- Check hierarchy level.
- Populate location traits.
- Populate record with message data.

DISTRICT MODIFY

- Check required fields.
- Check hierarchy level.
- Verify district record exists
- Get the existing region for district.
- Populate record with message data.

DISTRICT DELETE

- Check required fields.
- Verify the district exists.
- Check if district is currently used on the regionality tables.
- Populate record with message data.

LOCATION TRAIT CREATE

- Check required fields.
- Check hierarchy level.
- Populate location traits.
- Populate record with message data.

LOCATION TRAIT DELETE

- Check required fields.
- Check hierarchy level (area, region, district) exists.
- Populate location traits.
- Populate record with message data.

Note that location trait delete calls regionality_exists function.

Bulk or Single DML Module

All insert, update and delete SQL statements are located in the family package. This package is ORGANIZATIONAL_SQL. The private functions will call this package.

Filename: rmssub_xorghr_sqls/b.pls

```
RMSSUB_XORGHR_SQL.PERSIST_MESSAGE
           (O_error_message   IN OUT         VARCHAR2,
            I_hier_level      IN             VARCHAR2,
            I_org_hier_rec    IN             ORGANIZATIONAL_SQL.ORG-HIER_REC,
            I_message_type    IN             VARCHAR2,)
```

This function determines what type of database transaction it will call based on the message type.

CHAIN CREATE

- Create messages get added to the chain table.

CHAIN MODIFY

- Modify messages directly update the CHAIN and the STORE_HIERARCHY tables with changes.

CHAIN DELETE

- Delete messages directly remove chain records.

AREA CREATE

- Create messages get added to the area table.
- If location traits are also passed, location traits are added to the location area trait table.

AREA MODIFY

- Modify messages directly update the area and the STORE_HIERARCHY tables with changes.

AREA DELETE

- Delete messages directly remove area records.

REGION CREATE

- Create messages get added to the region table.
- Location traits from the area are added to the location region trait table.
- If location traits are also passed, location traits are added to the location region trait table.

REGION MODIFY

- Modify messages directly update the REGION and the STORE_HIERARCHY tables with changes.
- If the area was changed, then the old area’s location traits are removed from the location region trait table.
- If the area was changed, then the new area’s location traits are added to the location region trait table.

REGION DELETE

- Delete messages directly remove region records.

DISTRICT CREATE

- Create messages get added to the DISTRICT table.
- Location traits from the region are added to the location district trait table.
- If location traits are also passed, location traits are added to the location district trait table.

DISTRICT MODIFY

- Modify messages directly update the DISTRICT and the STORE_HIERARCHY tables with changes.
- If the region was changed, then the old region's location traits are removed from the location district trait table.
- If the region was changed, then the new region's location traits are added to the location district trait table.

DISTRICT DELETE

- Delete messages directly remove district records.

LOCATION TRAIT CREATE

- Add location trait records to the appropriate location trait table (area, region, district).
- Default the location trait records to each level below the passed in hierarchy (if region location traits are passed in, default those traits to all districts below that region and all stores below those districts).

LOCATION TRAIT DELETE

- Delete location trait records from the appropriate location trait table (area, region, district).
- Delete the location trait records from each level below the passed in hierarchy (if region location traits are passed in, delete those traits from all districts below that region and all stores below those districts).

Message DTD

Here are the filenames that correspond with each message type. Please consult the RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Type	Message Type Description	DTD Name
XOrgHrCre	External Create Organizational Hiearchy	XOrgHrDesc.dtd
XOrgHrLocTrtCre	External Create Organizational Hierarchy Location Trait	XOrgHrDesc.dtd
XOrgHrDel	External Delete Organizational Hiearchy	XOrgHrRef.dtd
XOrgHrLocTrtDel	External Delete Organizational Hiearchy Location Trait	XOrgHrRef.dtd
XOrgHrMod	External Modify Organizational Hiearchy	XOrgHrDesc.dtd

Design Assumptions

The REGIONALITY_HEAD table contains one record for each group/organizational hierarchy value that is defined for regionality. Regionality tables are used to define specific locations, suppliers, and/or departments that groups of users have responsibility for. These tables are not referenced within RMS, but can be used to customize reporting as well as on-line access if desired.

Tables

TABLE	SELECT	INSERT	UPDATE	DELETE
CHAIN	Yes	Yes	Yes	Yes
AREA	Yes	Yes	Yes	Yes
REGION	Yes	Yes	Yes	Yes
DISTRICT	Yes	Yes	Yes	Yes
LOC_AREA_TRAITS	Yes	Yes	No	Yes
LOC_REGION_TRAITS	Yes	Yes	No	Yes
LOC_DISTRICT_TRAITS	Yes	Yes	No	Yes
LOC_TRAITS_MATRIX	Yes	Yes	No	Yes
REGIONALITY_HEAD	Yes	No	No	No
REGIONALITY_DEPT	Yes	No	No	No
REGIONALITY_SUP_DEPT	Yes	No	No	No
REGIONALITY_SUP	Yes	No	No	No
REGIONALITY_TEMP	Yes	No	No	No

Payment Terms

Functional Area

Payment Terms

Business Overview

Payment terms are supplier-related financial arrangement information that is published to the Oracle Retail Integration Bus (RIB), along with the supplier and the supplier address, from the financial system. Payment terms are the terms established for paying a supplier (for example, 2.5% for 30 days, 3.5% for 15 days, 1.5% monthly, and so on). RMS subscribes to a payment terms message that is held on the RIB. After confirming the validity of the records enclosed within the message, RMS updates its tables with the information.

Data Flow:

An external system will publish a payment term, thereby placing the payment term information onto the RIB. RMS will subscribe to the payment term information as published from the RIB and place the information onto RMS tables depending upon the validity of the records enclosed within the message.

Message Structure:

The payment term message will consist of a payment term record header and detail.

The record will contain information about the payment term as a whole.

Package Impact

Filename: rmssub_ptrms/b.pls

Subscribing to a payment term message entails the use of one public consume procedure. This procedure corresponds to the type of activity that can be done to a payment term record (in this case create/update).

All of the following procedures exist within RMSSUB_PAYTERM.

```
CONSUME
       (O_status_code              OUT          VARCHAR2,
        O_error_message            OUT          VARCHAR2,
        I_message                  IN           RIB_OBJECT,
        I_message_type             IN           VARCHAR2)
```

This procedure initially checks that the passed in message type is a valid type for Terms messages. The valid message types for Terms messages are: paytermCre , paytermMod , paytermdtlCre and paytermdtlMod. If the message type is invalid, a status of “E” should be returned to the external system along with an appropriate error message informing the external system that the status is invalid.

If the message type is valid, the generic RIB_OBJECT will need to be downcast to the actual object using the Oracle’s treat function. There will be an object type that corresponds with each message type. If the downcast fails, a status of “E” should be returned to the external system along with an appropriate error message informing the external system that the object passed in is invalid.

If the downcast is successful, then consume needs to verify that the message passes all of the RMS business validation. It does not actually perform any validation itself; instead, it calls the RMSSUB_PAYTERM_VALIDATE.CHECK_MESSAGE function to determine whether the message is valid. This function is overloaded so simply passing the object in should be sufficient. If the message passed RMS business validation, then the function will return true, otherwise it will return false. If the message has failed RMS business validation, a status of "E" should be returned to the external system along with the error message returned from the CHECK_MESSAGE function.

Once the message has passed RMS business validation, it can be persisted to the RMS database. The consume function does not have to have any knowledge of how to persist the message to the database, it calls the RMSSUB_PAYTERM_SQL.PERSIST() function. This function is overloaded so simply passing the object should be sufficient. If the database persistence fails, the function will return false. A status of "E" should be returned to the external system along with the error message returned from the PERSIST() function.

Once the message has been successfully persisted, there is nothing more for the consume procedure to do. A success status, "S", should be returned to the external system indicating that the message has been successfully received and persisted to the RMS database.

Internal Procedure

```
HANDLE_ERROR
            (O_status_code          IN OUT        VARCHAR2,
            O_error_message         IN OUT        RTK_ERRORS.RTK_TEXT%TYPE,
            I_cause                 IN                  VARCHAR2,
            I_program               IN                  VARCHAR2)
```

This is the standard error handling function that wraps the API_LIBRARY.HANDLE_ERROR function.

The function consists of a call to API_LIBRARY.HANDLE_ERRORS. API_LIBRARY.HANDLE_ERRORS accepts a program name, the cause of the error and potentially an unparsed error message if one has been created through a call to SQL_LIB.CREATE_MESSAGE. The function uses these input variables to parse a complete error message and pass back a status, depending upon the message and error type, back up through the consume function and up to the RIB.

Business Validation Mode

Filename: rmssub_ptrmvals/b.pls

This function performs all business validation associated with Terms create and modify messages. It is important that the signature uses IN for the message and not IN OUT. When IN is used, the parameter is passed by reference. Passing by reference keeps the server from duplicating the memory allocation.

All of the following functions exist within RMSSUB_PAYTERM_VALIDATE.

```
CHECK_MESSAGE
           (O_error_message          OUT           RTK_ERRORS.RTK_TEXT%TYPE,
            O_dml_rec                OUT           TERMS_SQL.PAYTERM_REC,
            I_message                IN            RIB_PAYTERMDESC_REC,
            I_message_type           IN            VARCHAR2)
```

This function performs all business validation associated with create/modify messages and builds the order API record with default values for persistence in the payment terms related tables. Any invalid records passed at any time results in message failure.

This function calls CHECK_REQUIRED_FIELDS to make sure that all required fields are not NULL. CHECK_ENABLED is called to check for the validity of records with start_date_active and end_date_active with enabled flag. CHECK_TERMS_HEAD and CHECK_TERMS_DETAIL are called to check for header and detail records before inserting and updating TERMS_DEATIL table. Finally, the payment terms record used for DML is populated within the POPULATE_RECORD function and passed back to RMSSUB_PAYTERM.CONSUME.

Internal Functions

CHECK_REQUIRED_FIELDS

This function ensures that all required fields in the message are NOT NULL.

POPULATE_RECORDS

This function populates the payment terms output record with the values sent in the message.

CHECK_ENABLED

This function in a loop checks for start_date_active and end_date_active with the enabled_flag setting from RIB_MESSAGE. Declare cursor to retrieve vdate from table period and another cursor to retrieve start_date_active and end_date_active for the terms and terms_seq inputted from TERMS_DETAIL table. In a loop assign terms_seq to a local variable. Open cursor to retrieve start_date_active and end_date_Active from TERMS_DETAIL table. If terms_detail.start_date_active is after period.vdate and if enabled_flag from the rib message is ‘Y’, then raise program error. If end_date_active is < vdate and enabled_flag from the rib message is ‘Y’, then raise program error. If vdate > = start_date_active and <= end_date_active and enabled_flag is ‘N’ then raise a pogram error.

CHECK_TERMS_HEAD

This function will be responsible for checking TERMS_HEAD record before populating TERMS_DETAIL table for new terms record. Calling TERM_SQL.HEADER_EXISTS function will perform this check.

CHECK_TERMS_DETAIL

This function checks existence of terms_detail records before updating detail record. Calling TERM_SQL.DETAIL_EXISTS function will perform this check.

DML Module

Filename: rmssub_ptrm_sqls/b.pls

The following function exists within RMSSUB_PAYTERM_SQL.

```
PERSIST
          (O_error_message          OUT          RTK_ERRORS.RTK_TEXT%TYPE,
           I_message                IN           TERMS_SQL.PAYTERM_REC,
           I_message_type           IN           VARCHAR2)
```

Perform INSERT/UPDATE statements by calling the appropriate functions according to the message type and passing the data in a record to these functions.

For the message type indicating a header insert, populate the header record defined in the term_sql package and call the term_sql.insert_header function with this header record. For the message type indicating a header or a detail insert, call the term_sql.insert_detail function and pass to it the detail node from the message.

For the message type indicating a header update, populate the header record defined in the term_sql package and call the term_sql.update_header function with this header record. For the message type indicating a detail update, call the term_sql.update_detail function and pass to it the detail node from the message.

Message DTD

Here are the filenames that correspond with each message type. Please consult the RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
PayTermCre	Payment Terms Create Message	PayTermDesc.dtd
PayTermMod	Payment Terms Modify Message	PayTermDesc.dtd
PayTermDtlCre	Payment Terms Detail Create Message	PayTermDesc.dtd
PayTermDtlMod	Payment Terms Detail Modify Message	PayTermDesc.dtd

Design Assumptions

- One of the primary assumptions in the current API approach is that ease of code will outweigh performance considerations. It is hoped that the ‘trickle’ nature of the flow of data will decrease the need to dwell on performance issues and instead allow developers to code in the easiest and most straight forward manner.
- The adaptor is only setup to call stored procedures, not stored functions. Any public program then needs to be a procedure.

Tables

TABLE	SELECT	INSERT	UPDATE	DELETE
TERMS_DETAIL	Yes	Yes	Yes	No
TERMS_HEAD	Yes	Yes	Yes	No

Purchase Order (PO)

Functional Area

PO subscription

Design Overview

The sor_purchase_order_ind system option indicates whether RMS is the system of record for purchase orders. If RMS is the system of record, then RMS databases hold that information. If RMS is not the system of record, then that information is imported into RMS from external systems.

By setting the value of sor_purchase_order_ind to N (No), retailers cannot perform the following:

- Create, modify, or delete a purchase order.
- Maintain the items and locations on a purchase order online in RMS. Retailers can only view purchase orders.

When RMS is not the system of record, it may subscribe to the new purchase order subscription API. The subscription keeps RMS in sync with the external system that is responsible for maintaining purchase orders.

The following functions are not supported for externally generated, non-EDI purchase orders: open to buy (OTB), deals, bracket costing, order splitting, order scaling, contracts, letters of credit, harmonized tariff schedule (HTS), order revision, order expenses, order rounding, and required documents. It is assumed that externally generated non-EDI purchase orders are being interfaced expressly for the facilitation of inventory movement in RMS.

This API allows external systems to create, edit, and delete purchase orders within RMS. These transactions are performed immediately upon message receipt so success or failure can be communicated to the calling application.

Purchase order messages will be sent across the Oracle Retail Integration Bus (RIB). POs can be created, modified or deleted at the header or the detail level, each with its own message type.

Consume Module

Filename: rmssub_xorders/b.pls

```
RMSSUB_XORDER.CONSUME
               (O_status_code        IN OUT        VARCHAR2,
                O_error_message      IN OUT        RTK_ERRORS.RTK_TEXT%TYPE,
                I_message            IN            RIB_OBJECT,
                I_message_type       IN            VARCHAR2)
```

This procedure will need to initially ensure that the passed in message type is a valid type for purchase order messages. The valid message types for purchase order messages are listed in a section below.

If the message type is invalid, a status of “E” should be returned to the external system along with an appropriate error message informing the external system that the status is invalid.

If the message type is valid, the generic RIB_OBJECT needs to be downcast to the actual object using the Oracle's treat function. There will be an object type that corresponds with each message type. If the downcast fails, a status of "E" is returned to the external system along with an appropriate error message informing the external system that the object passed in is invalid.

If the downcast is successful, then consume needs to verify that the message passes all of the RMS business validation. It calls the RMSSUB_XORDER_VALIDATE.CHECK_MESSAGE function to determine whether the message is valid. If the message passed RMS business validation, then the function returns true, otherwise it returns false. If the message fails RMS business validation, a status of "E" is returned to the external system along with the error message returned from the CHECK_MESSAGE function.

Once the message has passed RMS business validation, it is persisted to the RMS database. It calls the RMSSUB_XORDER_SQL.PERSIST() function. If the database persistence fails, the function returns false. A status of "E" is returned to the external system along with the error message returned from the PERSIST() function.

Once the message has been successfully persisted, there is nothing more for the consume procedure to do. A success status, "S", is returned to the external system indicating that the message has been successfully received and persisted to the RMS database.

Business Validation Module

Filename: rmssub_xordervals/b.pls

It should be noted that some of the business validation is referential or involves uniqueness. This validation is handled automatically by the referential integrity constraints and the unique indexes implemented on the database.

```
RMSSUB_XORDER_VALIDATE.CHECK_MESSAGE
       (O_error_message      IN OUT              RTK_ERRORS.RTK_TEXT%TYPE,
        O_order_rec          OUT NOCOPY          ORDER_SQL.ORDER_REC,
        I_message            IN                  RIB_XORDERDESC_REC,
        I_message_type       IN                  VARCHAR2)
```

This overloaded function performs all business validation associated with create/modify messages and builds the order API record with default values for persistence in the order related tables. Any invalid records passed at any time results in message failure.

Like other APIs, the purchase order API expects a snapshot of the record on both a header modify and a detail modify message, instead of only the fields that are changed. For a detail create or a detail modify message, only the order number will be validated at the header level; all other header fields are ignored.

Defaulted fields that are not included in the message structure of the object must be populated in a package business record, ORDER_SQL.ORDER_REC. This record is used as input to the database DML functions in the persist package.

ORDER CREATE

- Check required fields on both header and detail nodes.
- Verify order number does NOT already exist.
- Verify attributes in the message header are correct.
- Verify attributes in the message detail are correct.
- Verify that item/supplier and item/supp/country exist for a non-pack item.

- Verify that item/supplier and item/supp/country exist for all components of a pack item.
- Create item/supplier and item/supp/country if they don't exist for a pack item.
- Create item/supp/country/loc if it does not exist for an item/location.
- Create item/loc relation if not already exist, including creating item_loc_soh, item_supp_country_loc, and price_hist records. If a pack item is involved, these records will be created for all component items.
- Populate record ORDER_REC with message data for both header and detail.

ORDER MODIFY

- Check required fields on the header node.
- Verify order number already exists.
- Verify attributes in the message header are correct.
- Verify attributes that cannot be modified are not changed.
- Update ORDLOC appropriately if closing or reinstating an order.
- Populate record ORDER_REC.ORDHEAD_ROW with message data.

ORDER DETAIL CREATE

- Check required fields on the detail node.
- Verify order number already exists.
- Verify order/item/loc does **not** already exist.
- Verify that item/supplier and item/supp/country exist for a non-pack item.
- Verify that item/supplier and item/supp/country exist for all components of a pack item.
- Create item/supplier and item/supp/country if they do not exist for a pack item.
- Create item/supp/country/loc if it does not exist for an item/location.
- Create item/loc relation if not already exists, including creating ITEM_LOC_SOH, ITEM_SUPP_COUNTRY_LOC, and PRICE_HIST records. If a pack item is involved, these records will be created for all component items.
- Populate record ORDER_REC.ORDLOCS and optionally, ORDER_REC.ORDSKUS with message data.

ORDER DETAIL MODIFY

- Check required fields on the detail node.
- Verify order/item/loc already exists.
- Verify attributes that cannot be modified are not changed.
- If order quantity is reduced, verify the new order quantity is not below what has already been received plus what is being shipped or expected.
- If the order line is cancelled or reinstated via the indicators, calculate the new quantity buckets.
- Populate record ORDER_REC.ORDLOCS and optionally, ORDER_REC.ORDSKUS with message data.

```
RMSSUB_XORDER_VALIDATE.CHECK_MESSAGE
               (O_error_message      IN OUT         RTK_ERRORS.RTK_TEXT%TYPE,
                O_order_rec          OUT NOCOPY     ORDER_SQL.ORDER_REC,
                I_message            IN             RIB_XORDERREF_REC,
                I_message_type       IN             VARCHAR2)
```

This overloaded function performs all business validation associated with delete messages and builds the order API record with default values for persistence in the order related tables. Any invalid records passed at any time results in message failure.

ORDER DELETE

- Check required fields.
- Verify order number already exists.
- Verify that order is not already shipped or received.
- Delete any allocations tied to the order
- Populate record ORDER_REC.ORDHEAD_ROW with the order number for delete.

ORDER DETAIL DELETE

- Check required fields.
- Verify order/item/loc already exists.
- Verify that order line is not already shipped or received.
- Delete any allocations tied to the order line.
- Populate record ORDER_REC.ORDLOCS with the order/item/location for delete.

Bulk or Single DML Module

Filename: rmssub_xorders/b.pls

All insert, update and delete SQL statements are located in package ORDER_SQL. The private functions call these packages.

```
RMSSUB_XORDER_SQL.PERSIST
               (O_error_message       IN OUT         RTK_ERRORS.RTK_TEXT%TYPE,
                I_order_rec           IN             ORDER_SQL.ORDER_REC,
                I_message_type        IN             VARCHAR2)
```

This function checks the message type to route the object to the appropriate internal functions that perform DML insert, update and delete processes.

ORDER CREATE

- Inserts records in the ORDHEAD, ORDSKU, ORDLOC tables

ORDER MODIFY

- Updates a record in the ORDHEAD table.

ORDER DELETE

- Delete an order from ORDHEAD, ORDSKU, ORDLOC tables.

ORDER DETAIL CREATE

- Inserts records in the ORDLOC and optionally, ORDSKU tables

ORDER DETAIL MODIFY

- Updates records in the ORDLOC and/or ORDSKU table.
- Also verify it doesn’t end up with an Approved order with 0 total order quantity.

ORDER DETAIL DELETE

- Delete records from ORDLOC and optionally, ORDSKU tables.
- Also verify it doesn’t end up with an Approved order with no detail or with 0 total order quantity.

Message DTD

Here are the filenames that correspond with each message type. Please consult the RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
XorderCre	Order Create Message	XOrderDesc.dtd
XorderMod	Order Modify Message	XOrderDesc.dtd
XorderDel	Order Delete Message	XOrderRef.dtd
XorderDtlCre	Order Detail Create Message	XOrderDesc.dtd
XorderDtlMod	Order Detail Modify Message	XOrderDesc.dtd
XorderDtlDel	Order Detail Delete Message	XOrderRef.dtd

Design Assumptions

Many ordering functionalities that are available on-line are not supported via this API. Triggers related to these functionalities should be turned off. Oracle Retail 11 deposit item functionality is not available in this API; that is to say a deposit contents item on the order does not automatically create the corresponding container item for the deposit item.

Tables

TABLE	SELECT	INSERT	UPDATE	DELETE
ORDHEAD	Yes	Yes	Yes	Yes
ORDSKU	Yes	Yes	Yes	Yes
ORDLOC	Yes	Yes	Yes	Yes
ITEM_SUPPLIER	Yes	Yes	No	No
ITEM_SUPP_COUNTRY	Yes	Yes	No	No
ITEM_SUPP_COUNTRY_LOC	Yes	Yes	No	No
ITEM_LOC	Yes	Yes	No	No
ITEM_LOC_SOH	Yes	Yes	No	No
PRICE_HIST	No	Yes	No	No
ITEM_ZONE_PRICE	Yes	No	No	No
ITEM_MASTER	Yes	No	No	No
PACKITEM_BREAKOUT	Yes	No	No	No
SHIPMENT	Yes	No	No	No
SHIPSKU	Yes	No	No	No
APPT_DETAIL	Yes	No	No	No

TABLE	SELECT	INSERT	UPDATE	DELETE
ALLOC_HEADER	Yes	No	No	Yes
ALLOC_DETAIL	Yes	No	No	Yes
STORE	Yes	No	No	No
WAREHOUSE	Yes	No	No	No
SUPS	Yes	No	No	No
DEPS	Yes	No	No	No
CURRENCIES	Yes	No	No	No
CURRENCY_RATES	Yes	No	No	No
TERMS	Yes	No	No	No
SYSTEM_OPTIONS	Yes	No	No	No
UNIT_OPTIONS	Yes	No	No	No
ADDR	Yes	No	No	No

Price Change

Functional Area

Price Change

Design Overview

RMS may subscribe to the price change subscription API. The price change subscription keeps RMS in sync with the external system that is responsible for maintaining price changes. The price change subscription updates prices for item/locations and item/zones that already exist in RMS. It does not create or delete item/locations or item/zones in RMS tables.

Price changes can be performed at the following levels of the organization hierarchy: chain, area, region, district, and store. Prices are updated for all stores within the location group unless marked as exceptions. Because warehouses are not part of the organization hierarchy, they are only impacted by price changes applied at the warehouse level.

The subscription does not create price change events; it updates the price or resets the clearance price of an item in real time.

The following rules are used to determine which stores are eligible for a price change:

- All stores in the location group based on the organization hierarchy (chain, area, region, district);
- Of the stores within the location group, those that have the same local currency as specified on the price change message;
- Of the stores with the same local currency, those in the same country as specified on the price change message;
- Of the remaining stores, those not found in the exception list.

RMS exposes an API that will allow external systems to update unit price within RMS.

This RMS API subscribes to external price change modify messages for the purpose of integrating external price changes maintained in an external system into RMS. It updates unit prices in RMS and writes to price history table.

At least one detail is required for a message to be valid.

Item, item parent, item parent/differentiator, organizational hierarchy, country, and currency can be used to maintain price changes. If price changes are created using the organizational hierarchy (area, region, and so on), then the country and currency filter the list of stores the price change will affect. Exception stores can be used to further limit stores impacted by price change.

If any locations (zones) do not exist on item-locations, those locations will not be processed and the message will not fail. When processing using either (or both) an item parent/organization hierarchy, records not found are not processed. However, if creating price changes by transactional level item/single location, then records not found on item-location will error out.

Price changes can update the single or multi unit retail, or both. Price changes will update location(s) if item is on clearance at the location(s). Non-sellable packs cannot be put on price changes.

When processing warehouse locations, those locations must be stockholding warehouses (virtual warehouse in multi-channel environment, physical in non-multi-channel environment).

This API only supports location (store) level zone pricing. A zone is equivalent to a location. In addition, this API only supports warehouse retail (RMS system_options.sor_pricing_ind = 'N').

Consume Module

Filename: rmssub_xpricechgs/b.pls

```
RMSSUB_XPRICECHG.CONSUME
               (O_status_code        IN OUT        VARCHAR2,
                O_error_message      IN OUT        RTK_ERRORS.RTK_TEXT%TYPE,
                I_message            IN            RIB_OBJECT,
                I_message_type       IN            VARCHAR2)
```

This procedure needs to initially ensure that the passed in message type is a valid type for Price change messages. There is only one valid message type for Price change messages, xpricechgmod. If the message type is invalid, a status of "E" should be returned to the external system along with an appropriate error message informing the external system that the status is invalid.

If the message type is valid, the generic RIB_OBJECT needs to be downcast to the actual object using the Oracle's treat function. If the downcast fails, a status of "E" should be returned to the external system along with an appropriate error message informing the external system that the object passed in is invalid.

If the downcast is successful, then consume needs to verify that the message passes all of the RMS business validation. It calls the RMSSUB_XPRICECHG_VALIDATE.CHECK_MESSAGE function to determine whether the message is valid. If the message passed RMS business validation, then the function will return true, otherwise it will return false. If the message has failed RMS business validation, a status of "E" should be returned to the external system along with the error message returned from the CHECK_MESSAGE function.

Once the message has passed RMS business validation, it can be persisted to the RMS database. It calls the RMSSUB_XPRICECHG_SQL.PERSIST() function. If the database persistence fails, the function returns false. A status of "E" should be returned to the external system along with the error message returned from the PERSIST() function.

Once the message has been successfully persisted, there is nothing more for the consume procedure to do. A success status, "S", should be returned to the external system indicating that the message has been successfully received and persisted to the RMS database.

Business Validation Module

Filename: rmssub_xpricechgvals/b.pls

It should be noted that some of the business validation is referential or involves uniqueness. This validation is handled automatically by the referential integrity constraints and the unique indexes implemented on the database.

```
RMSSUB_XPRICECHG_VALIDATE.CHECK_MESSAGE
       (O_error_message       IN OUT         RTK_ERRORS.RTK_TEXT%TYPE,
        O_price_change_rec    OUT            RMSSUB_XPRICECHG.PRICE_CHANGE_REC,
        I_message             IN             RIB_XPriceChgDesc)
```

This function performs all business validation associated with message and builds the price change record for persistence.

PRICE CHANGE MODIFY

- Check required fields.
- Validate passed in fields (currency, country, UOM, hierarchy level).
- Verify item is above transaction level and approved.
- If diff ids are passed in, verify they are valid for passed in item.
- Verify item passed in is not a non-sellable pack.
- Validate single and/or multi UOMs passed in are of the same UOM class as the standard UOM.
- Convert the new retail passed in from the UOM on the message to item's standard UOM.
- Determine price change type (single, multi or both).
- POPULATING RECORD
  - Retrieve item's transaction level children if the passed in item is a parent.
  - Retrieve all locations based on passed in organizational hierarchy type and value, currency and country, including those on clearance.
  - Exclude locations passed in as exception stores.
  - Build price change records.
- Populate record with message data.

Bulk or Single DML Module

Filename: rmssub_xpricechgsqls/b.pls

Insert, update and delete SQL statements are located in package PRICING_SQL. The private functions call these packages.

For non-pack items, this API calls STKLEDGR_PRICING_SQL package to write stock ledger for price change, which is optimized for performance. For a pack item, this API calls STKLEDGR_SQL package to write stock ledger.

```
RMSSUB_XPRICECHG_SQL.PERSIST
       (O_error_message       IN OUT         RTK_ERRORS.RTK_TEXT%TYPE,
        O_price_change_rec    OUT            RMSSUB_XPRICECHG.PRICE_CHANGE_REC)
```

PRICE CHANGE

- Update the unit retail on ITEM_LOC table for all item/locations if the single unit retail is changed and the item is not a pack. No update for multi-unit retail. Item locations that are currently on clearance will also be updated, with clearance indicator reset to ‘N’o.
- Update the unit retail(s) on ITEM_ZONE_PRICE table for all item/locations.
- Insert into price history all records that have item_loc relation, including sellable packs.
- TRAN_DATA is only inserted for transactional level items that have ITEM_LOC records. In addition, TRAN_DATA is only inserted if the old standard unit retail and the new standard unit retail are NOT the same. Do **not** insert tran_data if item/location’s stock_on_hand+in transit quantity is 0.
- For each TRAN_DATA record inserted, a SUP_DATA record will also be inserted

Message DTD

Here are the filenames that correspond with each message type. Please consult RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Type	Message Type Description	DTD Name
Xpricechgmod	External Price Change Modify	XPriceChgDesc.dtd

Design Assumptions

- Required fields are shown in the RIB documentation.
- Each price change message should be committed separately. A few global temporary tables are utilized in this API to help performance. Records on these global temporary tables will be deleted on commit. This API does **not** delete from these tables.
- This API only supports location (store) level zone pricing. Zone ID is equivalent to a location.
- This API only supports warehouse retail (RMS system_options.sor_pricing_ind = 'N'). As a result, warehouse retail is held on the ITEM_ZONE_PRICE and ITEM_LOC table in warehouse currency.
- If more than one price change of the same type (single or multi unit retail or both) occurs on an item/loc for a day, then price history is updated, as opposed to a new record being inserted for the second price change.
- In the current implementation of this API, POS_MODS are **not** written for price change.

Triggers impacting item/location tables should be turned off unless deemed necessary.

Tables

TABLE	SELECT	INSERT	UPDATE	DELETE
ITEM_ZONE_PRICE	Yes	No	Yes	No
ITEM_LOC	Yes	No	Yes	No
ITEM_LOC_SOH	Yes	No	No	No
STORE	Yes	No	No	No
WH	Yes	No	No	No
ITEM_MASTER	Yes	No	No	No
DIFF_GROUP_HEAD	Yes	No	No	No
DIFF_GROUP_DETAIL	Yes	No	No	No
CHAIN	Yes	No	No	No
AREA	Yes	No	No	No
REGION	Yes	No	No	No
DISTRICT	Yes	No	No	No
CURRENCIES	Yes	No	No	No
COUNTRY	Yes	No	No	No
ITEM_SUPP_COUNTRY_LOC	Yes	No	No	No
ITEM_SUPP_COUNTRY	Yes	No	No	No
PRICE_HIST	Yes	Yes	No	No

TABLE	SELECT	INSERT	UPDATE	DELETE
SUP_DATA	No	Yes	No	No
SUPS	Yes	No	No	No
TRAN_DATA	No	Yes	No	No
CLASS	Yes	No	No	No
VAT_ITEM	Yes	No	No	No
PERIOD	Yes	No	No	No
SYSTEM_OPTIONS	Yes	No	No	No
API_ITEM_TEMP	Yes	Yes	No	No
API_LOC_TEMP	Yes	Yes	No	No
API_ITEM_LOC_TEMP	Yes	Yes	No	No
API_PRICE_HIST_TEMP	Yes	Yes	No	No
API_PRICE_CHANGE_TEMP	Yes	Yes	No	No
API_ORIG_RETAIL_TEMP	Yes	Yes	No	No
API_VAT_TEMP	Yes	Yes	No	No

Receiving

Functional Area

Receipt subscription:

- Purchase Order Receiving
- Stock Order Receiving (including Transfers and Allocations)

Design Overview

RMS receives against purchase orders, transfers, and allocations. Transfers and allocations are collectively referred to as stock orders. The receipt subscription API processes carton-level receipts and a number of carton-level exceptions for stock orders receipts.

Purchase orders continue to be received only at the item level. If errors are encountered during purchase order receiving, the entire message is rejected and processing of the message stops.

Stock orders may be received at the bill of lading (BOL), carton, or item level. The following exceptions are automatically processed by the new stock order receiving package:

- Receiving against the wrong BOL.
- Receiving at a location which is a walk-through store for the intended location.
- Wrong store receiving.
- Unwanded cartons (those that have not been scanned).

Once RMS determines the appropriate receiving process for a carton, the shipment detail records are identified and existing line item level receiving is executed. The items are received into stock and transactions are updated.

Stock order may be received at the BOL (receiving the entire shipment without checking the details), carton (receiving the entire carton on SHIPSKU without checking the details), or item level. When an error is encountered during stock order receiving, an error record is created for the BOL, carton, or item in error. Processing continues for the remainder of the stock order receipt message. When the entire message has been processed, all of the error records are then handled. Error records are grouped together based on the type of error and a complete receipt message is created for each group. All errors will be collected in an error table, which will then be passed back to the RIB for further processing or hospitalization.

Carton-level Receiving

The process for handling carton level receipts is as follows:

1. RMS determines whether a message type contains a receipt or an appointment.
2. If a receipt, RMS determines whether the document type is purchase order (P), transfer (T), or allocation (A).
3. If a stock order (transfer or allocation), RMS determines whether the receipt is an item level receipt (SK) or a carton level receipt (BL).
4. If a carton level receipt, two scenarios are possible. The message may contain (a) a bill of lading number but no carton numbers or (b) a bill of lading and one or more carton numbers.
   - Bill of lading/no cartons: RMS receives all cartons associated with the BOL along with their contents (line items).
   - Bill of lading/with cartons: RMS receives only the specified cartons and their contents (line items).
5. The status of the cartons determines how the cartons/items are processed. The status may be Actual (A), Overage (O), or Dummy BOL (D).

   Actual (A)

   The cartons are received at the correct location against the correct bill of lading.

   Overage (O)

   The carton does not belong to the current BOL. RMS attempts to match the contents with the correct BOL.

   - If the carton belongs to a BOL at the given location, RMS receives the carton against the correct BOL at the given location.
   - If the carton belongs to a BOL at a related walk-through store, RMS receives the carton against the intended BOL at the intended location.
   - If the carton belongs to a BOL at an unrelated location, RMS uses the wrong store receiving process.

   Dummy BOL (D)

   Cartons were received under a dummy bill of lading (BOL) number. RMS attempts to match the contents with a valid BOL.

   - If the carton belongs to a valid BOL at the given location, RMS receives the carton against the intended BOL at the given location.
   - If the carton belongs to a valid BOL at a related walk-through store, RMS receives the carton against the intended BOL at the intended location.
   - If the carton belongs to a valid BOL at an unrelated location, RMS uses the wrong store receiving process.

   The wrong_st_receipt_ind system option controls whether wrong store receiving is available in RMS. The wrong_st_receipt_ind must be set to Y (Yes) to turn on this functionality. Wrong store receiving is done at the line item level. Inventory, average costs, and transactions for both the intended location and actual location are adjusted to accurately reflect the actual location of the items.

Doc Types

Receipts are processed based upon the document type indicator in the message. The indicator serves as a flag for RMSSUB_RECEIVE.CONSUME to use when calling the appropriate function that validates the data and writes the data to the base tables. The following are the document types and respective package and function names:

- **A –** for allocation. STOCK_ORDER_RCV_SQL.ALLOC_LINE_ITEM
- **P –** for purchase order. PO_RCV_SQL.PO_LINE_ITEM
- **T –** for transfer. STOCK_ORDER_RCV_SQL.TSF_LINE_ITEM

Blind Receipt Processing

A blind receipt is generated by an external application whenever a movement of goods is initiated by that application. RMS has no prior knowledge of blind receipts. RMS handles blind receipts when it runs STOCK_ORDER_RCV_SQL (transfers and allocations) or PO_RCV_SQL (purchase orders). If no appointment record exists on APPT_DETAIL, the respective function writes a record to the DOC_CLOSE_QUEUE table.

When a transfer, PO or allocation is received at a location, the external location (store or warehouse) will publish a receipt message to the RIB indicating that the stock has arrived. RMS will subscribe to the receipt message and update the appropriate tables, including shipment, transfer/allocation/purchase order, inventory and stock ledger.

For a stock order receiving (including transfers and allocations), the ownership of the goods moves to the receiving location at the time of shipment. As a result, financial transaction records are written for the goods shipped when RMS processes a BOL message. At the receiving time, financial transaction records will only need to be written for the overage receiving. In addition, the stock order receiving process also handles the situations where stock is received with no receipt, or if the stock is received at wrong stores, or if the item received is on a dummy carton.

The receipt message is a hierarchical message that can contain a series of receipts. Each receipt corresponds to a transfer or an allocation or a PO, and can contain carton or item details. Purchase orders are only received at the item level. Any errors encountered during purchase order receiving will cause the entire message to be rejected and processing of the message will stop.

Subscription Packages

Filename: rmssub_receivings/b.pls

```
RMSSUB_RECEIVING.CONSUME
               (O_status_code         IN OUT         VARCHAR2,
                O_error_message       IN OUT         VARCHAR2,
                I_message             IN             RIB_OBJECT,
                I_message_type        IN             VARCHAR2,
                O_rib_otbdesc_rec     OUT            RIB_OTBDESC_REC,
                O_rib_error_tbl       OUT            RIB_ERROR_TBL)
```

This procedure will make calls to receiving or appointment functions based on the value of I_message_type. If I_message type is RECEIPT_ADD or RECEIPT_UPD, then a call is made to RMSSUB_RECEIPT.CONSUME, casting the message as a RIB_RECEIPTDESC_REC. If I_message_type is APPOINT_HDR_ADD, APPOINT_HDR_UPD, APPOINT_HDR_DEL, APPOINT_DTL_ADD, APPOINT_DTL_UPD, or APPOINT_DTL_DEL, then a call is made to RMSSUB_APPOINT.CONSUME. This is the procedure called by the RIB.

```
RMSSUB_RECEIVING.HANDLE_ERRORS
               (O_status_code         IN OUT         VARCHAR2,
                IO_error_message      IN OUT         VARCHAR2,
                I_cause               IN             VARCHAR2,
                I_program             IN             VARCHAR2)
```

Standard error handling function that wraps the API_LIBRARY.HANDLE_ERROR function.

Filename: rmssub_receipts/b.pls

```
RMSSUB_RECEIPT.CONSUME
               O_status_code          IN OUT         VARCHAR2,
               O_error_message        IN OUT         VARCHAR2,
               I_rib_receiptdesc_rec  IN             RIB_RECEIPTDESC_REC,
               I_message_type         IN             VARCHAR2,
               O_rib_otbdesc_rec      OUT            RIB_OTBDESC_REC,
               O_rib_error_tbl        OUT            RIB_ERROR_TBL)
```

This function performs PO receiving and stock order receiving for each receipt in the message. Document type ‘P’ is for purchase order receiving, ‘A’ for allocation receiving, and ‘T’, ‘V’, ‘D’ for transfer receiving. All other document types are invalid.

Calls are made to ORDER_RCV_SQL.INIT_PO_ASN_LOC_GROUP, STOCK_ORDER_RCV_SQL.INIT_TSF_ALLOC_GROUP, and RMSSUB_RECEIPT_ERROR.INIT. These functions will initialize global variables and clean out cached info.

Loop through each receipt in the message. If the document type is 'P' (purchase order) then loop through the details of the receipt and call ORDER_RCV_SQL.PO_LINE_ITEM to receive the items on the PO. If the document type is 'T' (transfer) or 'A' (allocation) then call RMSSUB_STKORD_RECEIPT.CONSUME to handle stock order receiving. If the document type is not 'P', 'T', ‘D’, ‘V’ or 'A' then stop processing the message and return an error message.

After processing all receipts, call ORDER_RCV_SQL.FINISH_PO_ASN_LOC_GROUP, STOCK_ORDER_RCV_SQL.FINISH_TSF_ALLOC_GROUP, and RMSSUB_RECEIPT_ERROR.FINISH. These functions wrap up the processing for receiving and error logic.

If any records exist on the rib_otb_tbl returned by ORDER_RCV_SQL.FINISH_PO_ASN_LOC_GROUP, then create a rib_otbdesc_rec object and add the rib_otb_tbl to the object.

Filename: rmssub_stkord_receipts/b.pls

```
RMSSUB_STKORD_RECEIPT.CONSUME
               (O_status_code         IN OUT         VARCHAR2,
                O_error_message       IN OUT         RTK_ERRORS.RTK_TEXT%TYPE,
                I_appt                IN             APPT_HEAD.APPT%TYPE,
                I_rib_receipt_rec     IN             RIB_RECEIPT_REC)
```

This function will process stock order receiving for all records within the rib_receipt_rec passed in. First, this function calls RMSSUB_RECEIPT_ERROR.BEGIN_RECEIPT. This function will hold onto the header level information (appt_nbr and rib_receipt_rec), which may be used to create error objects.

Next, call RMSSUB_RECEIPT_VALIDATE.CHECK_RECEIPT, which will do validation at the receipt level. If the validation fails then reject the receipt by calling RMSSUB_RECEIPT_ERROR.ADD_ERROR.

The package will do carton-level receiving when receipt_type = 'BL', and item-level receiving when receipt_type = 'SK'.

There are two scenarios for carton-level receiving:

1. The rib_receipt_rec contains a bol_no and no cartons (no detail nodes). In this case call new function RMSSUB_STKORD_RECEIPT_VALIDATE.CHECK_BOL, which will do business level validation for the BOL. If the validation succeeds then call RMSSUB_STKORD_RECEIPT_SQL.PERSIST_BOL. If the validation fails then reject the BOL receipt by calling RMSSUB_RECEIPT_ERROR.ADD_ERROR.
2. The rib_receipt_rec contains a bol_no and 1 or more cartons (detail nodes). In this case, loop through each carton in the receipt and call new function RMSSUB_STKORD_RECEIPT_VALIDATE.CHECK_CARTON. This function will do business level validation for a carton. If the validation succeeds then call RMSSUB_STKORD_RECEIPT_SQL.PERSIST_CARTON. If the validation fails because the carton is a duplicate (check the returned validation_code), then just skip over the call to PERSIST_CARTON and continue. Duplicates should be ignored with no error. If the validation fails for any other reason then reject the carton by calling RMSSUB_RECEIPT_ERROR.ADD_ERROR.

Item (SKU) Level Receiving:

If the receipt is item-level ('SK') then loop through the detail records and call new function RMSSUB_STKORD_RECEIPT_VALIDATE.CHECK_ITEM, which will do business level validation for the item details. If the validation succeeds then call RMSSUB_STKORD_RECEIPT_SQL.PERSIST_LINE_ITEM to execute existing line item receiving package calls. If the validation fails then reject the item by calling RMSSUB_RECEIPT_ERROR.ADD_ERROR.

When all details for the receipt have been processed, or if the entire receipt itself is rejected, then call RMSSUB_RECEIPT_ERROR.END_RECEIPT. This function will group all similar errors and create the appropriate error objects.

If a break to sell sellable item is on the message, it calls CHECK_ITEM and GET_ORDERABLE_ITEMS to convert the sellable to its orderable items. For a break to sell item, the orderable items are on the transfers, allocations, shipment, inventory and stock ledger.

Filename: rmssub_stkord_rct_vals/b.pls

```
RMSSUB_STKORD_RECEIPT_VALIDATE.CHECK_RECEIPT
               (O_error_message      IN OUT         RTK_ERRORS.RTK_TEXT%TYPE,
                O_valid              OUT            BOOLEAN,
                O_validation_code    OUT            VARCHAR2,
                I_rib_receipt_rec    IN             RIB_RECEIPT_REC)
```

This function will perform business validation for a receipt. If any of the validations fail then populate O_validation_error with the specified error code and set O_valid = FALSE. Otherwise, leave O_validation_error as NULL and set O_valid = TRUE.

```
RMSSUB_STKORD_RECEIPT_VALIDATE.CHECK_BOL
            (O_error_message          IN OUT  RTK_ERRORS.RTK_TEXT%TYPE,
             O_valid                  IN OUT  BOOLEAN,
             O_validation_code        IN OUT  VARCHAR2,
             O_shipment               IN OUT  SHIPMENT.SHIPMENT%TYPE,
             O_item_table             IN OUT  STOCK_ORDER_RCV_SQL.ITEM_TAB,
             O_qty_expected_table     IN OUT  STOCK_ORDER_RCV_SQL.QTY_TAB,
             O_inv_status_table       IN OUT  STOCK_ORDER_RCV_SQL.INV_STATUS_TAB,
             O_carton_table           IN OUT  STOCK_ORDER_RCV_SQL.CARTON_TAB,
             O_distro_no_table        IN OUT  STOCK_ORDER_RCV_SQL.DISTRO_NO_TAB,
             O_tampered_ind_table     IN OUT  STOCK_ORDER_RCV_SQL.TAMPERED_IND_TAB,
             I_bol_no                 IN      SHIPMENT.BOL_NO%TYPE,
             I_to_loc                 IN      SHIPMENT.TO_LOC%TYPE)
```

This function will perform business validation for receipts using BOL-level receiving. During validation this function selects data from the SHIPMENT and SHIPSKU tables and passes this information out through the parameters. This is done so that these tables do not have to be hit again during the receiving (persist) process. If any of the validations fail then populate O_validation_error with the specified error code and set O_valid = FALSE. Otherwise, leave O_validation_error as NULL and set O_valid = TRUE.

```
RMSSUB_STKORD_RECEIPT_VALIDATE.CHECK_CARTON
          (O_error_message              IN OUT  RTK_ERRORS.RTK_TEXT%TYPE,
           O_valid                      IN OUT  BOOLEAN,
           O_validation_code            IN OUT  VARCHAR2,
           O_ctn_shipment               IN OUT  SHIPMENT.SHIPMENT%TYPE,
           O_ctn_to_loc                 IN OUT  SHIPMENT.TO_LOC%TYPE,
           O_ctn_bol_no                 IN OUT  SHIPMENT.BOL_NO%TYPE,
           O_item_table                 IN OUT  STOCK_ORDER_RCV_SQL.ITEM_TAB,
           O_qty_expected_table         IN OUT  STOCK_ORDER_RCV_SQL.QTY_TAB,
           O_inv_status_table           IN OUT  STOCK_ORDER_RCV_SQL.INV_STATUS_TAB,
           O_carton_table               IN OUT  STOCK_ORDER_RCV_SQL.CARTON_TAB,
           O_distro_no_table            IN OUT  STOCK_ORDER_RCV_SQL.DISTRO_NO_TAB,
           O_tampered_ind_table         IN OUT  STOCK_ORDER_RCV_SQL.TAMPERED_IND_TAB,
           O_wrong_store_ind            IN OUT  VARCHAR2,
           O_wrong_store                IN OUT  SHIPMENT.TO_LOC%TYPE,
           I_bol_no                     IN      SHIPMENT.BOL_NO%TYPE,
           I_to_loc                     IN      SHIPMENT.TO_LOC%TYPE,
           I_from_loc                   IN      SHIPMENT.FROM_LOC%TYPE,
           I_from_loc_type              IN      SHIPMENT.FROM_LOC_TYPE%TYPE,
           I_rib_receiptcartondtl_rec   IN      RIB_RECEIPTCARTONDTL_REC)
```

This function will perform business validation for receipts using carton-level receiving. Based on the carton status, a carton can be received to the intended store only, or as a dummy carton or to the walk-through store of the intended store.

During validation this function selects data from SHIPMENT and SHIPSKU tables and passes this information out through the parameters. This is done so that these tables do not have to be hit again during the receiving (persist) process. If any of the validations fail then populate O_validation_error with the specified error code and set O_valid = FALSE. Otherwise, leave O_validation_error as NULL and set O_valid = TRUE.

```
RMSSUB_STKORD_RECEIPT_VALIDATE.CHECK_ITEM
               (O_error_message       IN OUT         RTK_ERRORS.RTK_TEXT%TYPE,
                O_valid               OUT            BOOLEAN,
                O_validation_code     OUT            VARCHAR2,
                I_distro_no           IN             SHIPSKU.DISTRO_NO%TYPE,
                I_dummy_carton_ind    IN             VARCHAR2)
```

This function will perform business validation for item details. If any of the validations fail then populate O_validation_error with the specified error code and set O_valid = FALSE. Otherwise, leave O_validation_error as NULL and set O_valid = TRUE.

```
RMSSUB_STKORD_RECEIPT_SQL.PERSIST_BOL
(O_error_message              IN OUT         RTK_ERRORS.RTK_TEXT%TYPE,
 I_appt                       IN             APPT_HEAD.APPT%TYPE,
 I_doc_type                   IN             APPT_DETAIL.DOC_TYPE%TYPE,
 I_shipment                   IN             SHIPMENT.SHIPMENT%TYPE,
 I_to_loc                     IN             SHIPMENT.TO_LOC%TYPE,
 I_bol_no                     IN             SHIPMENT.BOL_NO%TYPE,
 I_item_table                 IN             STOCK_ORDER_RCV_SQL.ITEM_TAB,
 I_qty_expected_table         IN             STOCK_ORDER_RCV_SQL.QTY_TAB,
 I_inv_status_table           IN             STOCK_ORDER_RCV_SQL.INV_STATUS_TAB,
 I_carton_table               IN             STOCK_ORDER_RCV_SQL.CARTON_TAB,
 I_distro_no_table            IN             STOCK_ORDER_RCV_SQL.DISTRO_NO_TAB,
 I_tampered_ind_table         IN             STOCK_ORDER_RCV_SQL.TAMPERED_IND_TAB)
```

This function calls STOCK_ORDER_RCV_SQL.TSF_BOL_CARTON (for transfers) and STOCK_ORDER_RCV_SQL.ALLOC_BOL_CARTON (for allocations) to perform BOL level receiving.

```
RMSSUB_STKORD_RECEIPT_SQL.PERSIST_CARTON
(O_error_message              IN OUT  RTK_ERRORS.RTK_TEXT%TYPE,
 I_appt                       IN      APPT_HEAD.APPT%TYPE,
 I_doc_type                   IN      APPT_DETAIL.DOC_TYPE%TYPE,
 I_shipment                   IN      SHIPMENT.SHIPMENT%TYPE,
 I_to_loc                     IN      SHIPMENT.TO_LOC%TYPE,
 I_bol_no                     IN      SHIPMENT.BOL_NO%TYPE,
 I_receipt_no                 IN      APPT_DETAIL.RECEIPT_NO%TYPE,
 I_disposition                IN      INV_STATUS_CODES.INV_STATUS_CODE%TYPE,
 I_receipt_date               IN      SHIPMENT.RECEIVE_DATE%TYPE,
 I_item_table                 IN      STOCK_ORDER_RCV_SQL.ITEM_TAB,
 I_qty_expected_table         IN      STOCK_ORDER_RCV_SQL.QTY_TAB,
 I_weight                     IN      ITEM_LOC_SOH.AVERAGE_WEIGHT%TYPE,
 I_weight_uom                 IN      UOM_CLASS.UOM%TYPE,
 I_inv_status_table           IN      STOCK_ORDER_RCV_SQL.INV_STATUS_TAB,
 I_carton_table               IN      STOCK_ORDER_RCV_SQL.CARTON_TAB,
 I_distro_no_table            IN      STOCK_ORDER_RCV_SQL.DISTRO_NO_TAB,
 I_tampered_ind_table         IN      STOCK_ORDER_RCV_SQL.TAMPERED_IND_TAB,
 I_wrong_store_ind            IN      VARCHAR2,
 I_wrong_store                IN      SHIPMENT.TO_LOC%TYPE)
```

This function calls STOCK_ORDER_RCV_SQL.TSF_BOL_CARTON (for transfers) and STOCK_ORDER_RCV_SQL.ALLOC_BOL_CARTON (for allocations) to perform carton level receiving.

```
RMSSUB_STKORD_RECEIPT_SQL.PERSIST_LINE_ITEM
               (O_error_message      IN OUT         RTK_ERRORS.RTK_TEXT%TYPE,
                I_location           IN             SHIPMENT.TO_LOC%TYPE,
                I_bol_no             IN             SHIPMENT.BOL_NO%TYPE,
                I_distro_no          IN             SHIPSKU.DISTRO_NO%TYPE,
                I_distro_type        IN             VARCHAR2,
                I_appt               IN             APPT_HEAD.APPT%TYPE,
                I_rib_receiptdtl_rec IN             RIB_RECEIPTDTL_REC)
```

This function calls STOCK_ORDER_RCV_SQL.TSF_LINE_ITEM (for transfers) and STOCK_ORDER_RCV_SQL.ALLOC_LINE_ITEM (for allocations) to perform item level receiving.

Filename: rmssub_receipt_errors/b.pls

For each item or carton found to be in error during the receiving process, an error record will be created. When all details for a receipt have been processed, the error records for that receipt will be grouped by the error type. Error objects will be collected in an error table, which will be passed back to the RIB for additional processing. This type of error handling will allow all valid records to be processed even when an invalid record is encountered.

```
RMSSUB_RECEIPT_ERROR.INIT
               (O_error_message      IN OUT  RTK_ERRORS.RTK_TEXT%TYPE)
```

This new function will initialize variables for error processing. It will be called in the 'init' section of the RMSSUB_RECEIPT.CONSUME() function.

```
RMSSUB_RECEIPT_ERROR.BEGIN_RECEIPT
               (O_error_message      IN OUT         RTK_ERRORS.RTK_TEXT%TYPE,
                I_appt_nbr           IN             APPT_HEAD.APPT%TYPE,
                I_rib_receipt_rec    IN             RIB_RECEIPT_REC)
```

This new function will be called once for each receipt within RMSSUB_STKORD_RECEIPT.CONSUME(). It will copy the header information into the package level variables. This information will be used when an error record is created.

```
RMSSUB_RECEIPT_ERROR.ADD_ERROR
               (O_error_message      IN OUT         RTK_ERRORS.RTK_TEXT%TYPE,
                I_error_type         IN             VARCHAR2,
                I_error_code         IN             VARCHAR2,
                I_error_desc         IN             VARCHAR2,
                I_error_level        IN             VARCHAR2,
                I_error_object       IN             RIB_OBJECT)
```

Used whenever an item or carton error occurs within the stock order receiving process. All calls to this function should occur within RMSSUB_STKORD_RECEIPT.CONSUME.

Parameter explanation:

- O_error_message: any error message created if this function fails (EXCEPTION).
- I_error_type: either 'BL' for business logic process error, or 'SY' for system error. Currently, 'BL' type errors will be limited to BOL/carton level business validation errors.
- I_error_code: a specific code to identify why/how the error occurred.
- I_error_desc: text description of the error.
- I_error_level: lets the package know how to cast the I_detail_rec. Valid values are 'RECEIPT', 'BOL', 'CARTON', 'ITEM'.
- I_detail_rec: record which is in error. May be a rib_receipt_rec (RECEIPT or BOL level), rib_receiptdtl_rec (ITEM level), or rib_receiptcartondtl_rec (CARTON level). This value will be cast based on I_error_level.

This function will create a new error record based on the error level passed in (casting the I_error_object appropriately). If the error level is RECEIPT or BOL, then a rib_receipt_rec is created. If the error level is CARTON, a rib_receiptcartondtl_rec is created. If error level is ITEM, a rib_receiptdtl_rec is created. After creating this error record, add it to the table of error records.

```
RMSSUB_RECEIPT_ERROR.END_RECEIPT
              (O_error_message  IN OUT  RTK_ERRORS.RTK_TEXT%TYPE)
```

This function will be called from RMSSUB_STKORD_RECEIPT.CONSUME when all details of a receipt have been processed. It will take all of the error records for this receipt and group them according to the type of error. It will then create an error object for each error type, adding detail nodes for each error record. When this is finished, it will add all of the error records to the error table.

```
RMSSUB_RECEIPT_ERROR.FINISH
              (O_error_message      IN OUT         RTK_ERRORS.RTK_TEXT%TYPE,
               O_rib_error_tbl      OUT            RIB_ERROR_TBL)
```

If any errors exist on the package level error table then copy the error table into the output parameter (O_rib_error_tbl), which in turn gets passed out to the RIB for further processing. This function will be called in the 'finish' section of the RMSSUB_STKORD_RECEIPT.CONSUME function.

Filename: stkordrcvs/b.pls

```
STOCK_ORDER_RCV_SQL.TSF_BOL_CARTON
(O_error_message       IN OUT         RTK_ERRORS.RTK_TEXT%TYPE,
 I_appt                IN             APPT_HEAD.APPT%TYPE,
 I_shipment            IN             SHIPMENT.SHIPMENT%TYPE,
 I_to_loc              IN             SHIPMENT.TO_LOC%TYPE,
 I_bol_no              IN             SHIPMENT.BOL_NO%TYPE,
 I_receipt_no          IN             APPT_DETAIL.RECEIPT_NO%TYPE,
 I_disposition         IN             INV_STATUS_CODES.INV_STATUS_CODE%TYPE,
 I_tran_date           IN             PERIOD.VDATE%TYPE,
 I_item_table          IN             ITEM_TAB,
 I_qty_expected_table  IN             QTY_TAB,
 I_weight              IN             ITEM_LOC_SOH.AVERAGE_WEIGHT%TYPE,
 I_weight_uom          IN             UOM_CLASS.UOM%TYPE,
 I_inv_status_table    IN             INV_STATUS_TAB,
 I_carton_table        IN             CARTON_TAB,
 I_distro_no_table     IN             DISTRO_NO_TAB,
 I_tampered_ind_table  IN             TAMPERED_IND_TAB,
 I_wrong_store_ind     IN             VARCHAR2,
 I_wrong_store         IN             SHIPMENT.TO_LOC%TYPE)
```

This function performs the BOL or carton level receiving for a transfer. It does the following:

- Update shipment to received status along with the received date.
- For each item on the SHIPSKU, build an API record for transferring the item. An orderable but non-sellable and non-inventory item cannot be transferred. The message will contain physical locations, but a transfer created in RMS (non-'EG' type) will contain virtual locations only. Convert the physical locations to virtual locations if necessary.
- Because an externally generated transfer (type 'EG') holds physical locations on TSFHEAD, and physical warehouses do **not** have transfer entities, this API does **not** support the receiving of an externally generated warehouse to warehouse transfer when system option INTERCOMPANY_TSF_IND is 'Y'. However, it does allow store to warehouse 'EG' transfer, because it is assumed that store is sending merchandise to the virtual warehouse within the same channel, hence the same transfer entity.
- When receiving a transfer to a finisher location, all stock will be received into the available bucket regardless of the inventory disposition on the message.
- When system option WRONG_ST_RECEIPT is 'Y', stock can be received at a store not originally intended. Inventory and stock ledger will be adjusted for both the intended and the actual receiving store.
- Update received quantity on TSFDETAIL. If it is a wrong store receiving, update the reconciled quantity on TSFDETAIL.
- Update received quantity and received weight on SHIPSKU. If SHIPSKU is not found, create a new receipt for that.
- For an 'EG' type of transfer, distribute the received quantity among the virtual locations of the physical location based on SHIPMENT_INV_FLOW, and update the received quantity on SHIPMENT_INV_FLOW.
- For an 'MRT' type of transfer, update received quantity on MRT_ITEM_LOC.
- Update APPT_DETAIL if appointment exists for the transfer detail; otherwise, insert into DOC_CLOSE_QUEUE.
- Call DETAIL_PROCESSING to perform the bulk of the transfer receiving logic, including moving inventory from the in transit to the stock on bucket for the receiving location. For overage receiving, adjust stock on hand for both the sending and receiving locations, adjust av_cost for the receiving location and write stock ledger.

```
STOCK_ORDER_RCV_SQL.TSF_LINE_ITEM
       (O_error_message       IN OUT  RTK_ERRORS.RTK_TEXT%TYPE,
        I_loc                 IN      ITEM_LOC.LOC%TYPE,
        I_item                IN      ITEM_MASTER.ITEM%TYPE,
        I_qty                 IN      TRAN_DATA.UNITS%TYPE,
        I_weight              IN      ITEM_LOC_SOH.AVERAGE_WEIGHT%TYPE,
        I_weight_uom          IN      UOM_CLASS.UOM%TYPE,
        I_transaction_type    IN      VARCHAR2,
        I_tran_date           IN      PERIOD.VDATE%TYPE,
        I_receipt_number      IN      APPT_DETAIL.RECEIPT_NO%TYPE,
        I_bol_no              IN      SHIPMENT.BOL_NO%TYPE,
        I_appt                IN      APPT_HEAD.APPT%TYPE,
        I_carton              IN      SHIPSKU.CARTON%TYPE,
        I_distro_type         IN      VARCHAR2,
        I_distro_number       IN      TSFHEAD.TSF_NO%TYPE,
        I_disp                IN      INV_STATUS_CODES.INV_STATUS_CODE%TYPE,
```

```
        I_tampered_ind        IN      SHIPSKU.TAMPERED_IND%TYPE,
        I_dummy_carton_ind    IN      SYSTEM_OPTIONS.DUMMY_CARTON_IND%TYPE)
```

Similar to TSF_BOL_CARTON, this function performs transfer receiving for one line item. In addition, if system_options DUMMY_CARTON_IND = 'Y' and the item is indicated as a dummy carton on the message, it will write staging records to the DUMMY_CARTON_STAGE table. The actual matching and receiving of dummy carton transfers will be performed during the batch cycle via dummyctn.pc.

```
STOCK_ORDER_RCV_SQL.ALLOC_BOL_CARTON
    (O_error_message       IN OUT     RTK_ERRORS.RTK_TEXT%TYPE,
     I_appt                IN         APPT_HEAD.APPT%TYPE,
     I_shipment            IN         SHIPMENT.SHIPMENT%TYPE,
     I_to_loc              IN         SHIPMENT.TO_LOC%TYPE,
     I_bol_no              IN         SHIPMENT.BOL_NO%TYPE,
     I_receipt_no          IN         APPT_DETAIL.RECEIPT_NO%TYPE,
     I_disposition         IN         INV_STATUS_CODES.INV_STATUS_CODE%TYPE,
     I_tran_date           IN         PERIOD.VDATE%TYPE,
     I_item_table          IN         ITEM_TAB,
     I_qty_expected_table  IN         QTY_TAB,
     I_weight              IN         ITEM_LOC_SOH.AVERAGE_WEIGHT%TYPE,
     I_weight_uom          IN         UOM_CLASS.UOM%TYPE,
     I_inv_status_table    IN         INV_STATUS_TAB,
     I_carton_table        IN         CARTON_TAB,
     I_distro_no_table     IN         DISTRO_NO_TAB,
     I_tampered_ind_table  IN         TAMPERED_IND_TAB,
     I_wrong_store_ind     IN         VARCHAR2,
     I_wrong_store         IN         SHIPMENT.TO_LOC%TYPE)
```

This function performs the BOL or carton level receiving for an allocation. It does the following:

- Update shipment to received status along with the received date.
- For each item on the SHIPSKU, build an API record for allocating the item. An orderable but non-sellable and non-inventory item cannot be allocated.
- Validate that item is on the allocation.
- When system option WRONG_ST_RECEIPT is 'Y', stock can be received at a store not originally intended. Inventory and stock ledger will be adjusted for both the intended and the actual receiving store.
- Validate that ALLOC_DETAIL exists. Update received quantity on ALLOC_DETAIL. If it is a wrong store receiving, update the reconciled quantity on ALLOC_DETAIL.
- Update received quantity and received weight on SHIPSKU. If SHIPSKU is not found, create a new receipt for that.
- Update APPT_DETAIL if appointment exists for the allocation detail; otherwise, insert into DOC_CLOSE_QUEUE.
- Call DETAIL_PROCESSING to perform the bulk of the allocation receiving logic, including moving inventory from the in transit to the stock on bucket for the receiving location. For overage receiving, adjust stock on hand for both the sending and receiving locations, adjust av_cost for the receiving location and write stock ledger.

```
STOCK_ORDER_RCV_SQL.ALLOC_LINE_ITEM
       (O_error_message       IN OUT  RTK_ERRORS.RTK_TEXT%TYPE,
        I_loc                 IN      ITEM_LOC.LOC%TYPE,
        I_item                IN      ITEM_MASTER.ITEM%TYPE,
        I_qty                 IN      TRAN_DATA.UNITS%TYPE,
        I_weight              IN      ITEM_LOC_SOH.AVERAGE_WEIGHT%TYPE,
        I_weight_uom          IN      UOM_CLASS.UOM%TYPE,
        I_transaction_type    IN      VARCHAR2,
        I_tran_date           IN      PERIOD.VDATE%TYPE,
        I_receipt_number      IN      APPT_DETAIL.RECEIPT_NO%TYPE,
        I_bol_no              IN      SHIPMENT.BOL_NO%TYPE,
        I_appt                IN      APPT_HEAD.APPT%TYPE,
        I_carton              IN      SHIPSKU.CARTON%TYPE,
        I_distro_type         IN      VARCHAR2,
        I_distro_number       IN      ALLOC_HEADER.ALLOC_NO%TYPE,
        I_disp                IN      INV_STATUS_CODES.INV_STATUS_CODE%TYPE,
        I_tampered_ind        IN      SHIPSKU.TAMPERED_IND%TYPE,
        I_dummy_carton_ind    IN      SYSTEM_OPTIONS.DUMMY_CARTON_IND%TYPE)
```

Similar to ALLOC_BOL_CARTON, this function performs allocation receiving for one line item. In addition, if system_options DUMMY_CARTON_IND = 'Y' and the item is indicated as a dummy carton on the message, it will write staging records to the DUMMY_CARTON_STAGE table. The actual matching and receiving of dummy carton allocations will be performed during the batch cycle via dummyctn.pc.

```
STOCK_ORDER_RCV_SQL.INIT_TSF_ALLOC_GROUP
              (O_error_message  IN OUT  RTK_ERRORS.RTK_TEXT%TYPE)
```

For performance reasons, bulk processing is used for stock order receiving. This function initializes global variables for bulk processing and populates system options.

```
STOCK_ORDER_RCV_SQL.FINISH_TSF_ALLOC_GROUP
              (O_error_message  IN OUT  RTK_ERRORS.RTK_TEXT%TYPE)
```

For performance reasons, bulk processing is used for stock order receiving. This function bulk updates APPT_DETAIL, bulk updates DOC_CLOSE_QUEUE and TRAN_DATA.

Filename: ordrcvs/b.pls

```
ORDER_RCV_SQL.PO_LINE_ITEM
       (O_error_message   IN OUT     rtk_errors.rtk_text%TYPE,
        I_loc             IN         item_loc.loc%TYPE,
        I_order_no        IN         ordhead.order_no%TYPE,
        I_item            IN         item_master.item%TYPE,
        I_qty             IN         tran_data.units%TYPE,
        I_tran_type       IN         VARCHAR2,
        I_tran_date       IN         DATE,
        I_receipt_number  IN         appt_detail.receipt_no%TYPE,
        I_asn             IN         shipment.asn%TYPE,
        I_appt            IN         appt_head.appt%TYPE,
        I_carton          IN         shipsku.carton%TYPE,
        I_distro_type     IN         VARCHAR2,
        I_distro_number   IN         alloc_header.alloc_no%TYPE,
        I_destination     IN         alloc_detail.to_loc%TYPE,
        I_disp            IN         inv_status_codes.inv_status_code%TYPE,
        I_unit_cost       IN         ordloc.unit_cost%TYPE,
        I_shipped_qty     IN         shipsku.qty_expected%TYPE,
        I_weight          IN         item_loc_soh.average_weight%TYPE,
        I_weight_uom      IN         UOM_CLASS.UOM%TYPE,
        I_online_ind      IN         VARCHAR2)
```

This function is called once for each PO line item received. It validates input and calls RCV_LINE_ITEM for each item/location.

- In a multi-channel environment, if the PO received is a cross-dock PO to a warehouse, an allocation must exist for the PO/allocation/item/warehouse combination. The message will contain a physical warehouse, whereas ALLOC_HEADER will contain a virtual warehouse.
- In a multi-channel environment, if the item is received to a physical warehouse, then this function will call the distribution logic to determine each item/virtual warehouse/quantity, and call RCV_LINE_ITEM for each of these combinations.
- If a simple pack catch weight item is received, it also updates SHIPSKU weight received and weight received UOM.

```
ORDER_RCV_SQL.RCV_LINE_ITEM
       (O_error_message   IN OUT     rtk_errors.rtk_text%TYPE,
        I_phy_loc         IN         item_loc.loc%TYPE,
        I_loc             IN         item_loc.loc%TYPE,
        I_loc_type        IN         item_loc.loc_type%TYPE,
        I_order_no        IN         ordhead.order_no%TYPE,
        I_item            IN         item_master.item%TYPE,
        I_qty             IN         tran_data.units%TYPE,
        I_tran_type       IN         VARCHAR2,
        I_tran_date       IN         DATE,
        I_receipt_number  IN         appt_detail.receipt_no%TYPE,
        I_asn             IN         shipment.asn%TYPE,
        I_appt            IN         appt_head.appt%TYPE,
        I_carton          IN         shipsku.carton%TYPE,
        I_distro_type     IN         VARCHAR2,
        I_distro_number   IN         tsfhead.tsf_no%TYPE,
        I_destination     IN         alloc_detail.to_loc%TYPE,
        I_disp            IN         inv_status_codes.inv_status_code%TYPE,
        I_unit_cost       IN         ordloc.unit_cost%TYPE,
        I_shipped_qty     IN         shipsku.qty_expected%TYPE,
        I_weight          IN         item_loc_soh.average_weight%TYPE,
        I_weight_uom      IN         UOM_CLASS.UOM%TYPE,
        I_online_ind      IN         VARCHAR2)
```

This function is called for each item/location combination. It validates input and performs PO receiving logic for each item.

- Receiving (tran_type = ‘R’) must be against a valid approved order; adjustment (tran_type = ‘A’) must be against a valid approved or closed order.
- Item on the message may be a referential item. Get its transaction level item.
- An orderable, but non-sellable and non-inventory item cannot be received.
- For a deposit content item, its container item is also received and added to the order if not already on the order.
- Insert or update ORDLOC for quantity received.
- Update APPT_DETAIL if appointment exists; otherwise, insert into DOC_CLOSE_QUEUE.
- Insert or update SHIPMENT to received status.
- Insert or update SHIPSKU for received quantity. If SHIPSKU.QTY_RECEIVED is updated, also update INVC_MATCH_WKSHT.MATCH_TO_QTY.
- If no deals exist for this order/item/loc, then call INVC_SQL.UPDATE_INVOICE to perform invoice matching logic.

- Update average cost and stock on hand for the stock received. If a pack is on the order, the updates are performed for the component items.
- Write TRAN_DATA records (tran code 20) for the stock received. If a pack is on the order, TRAN_DATA records are written for the component items.
- Write SUP_DATA.
- Request tickets to be printed if location is a store.
- If this is an adjustment to a closed order, set status back to 'A'pproved.

Message DTD

Here are the filenames that correspond with each message type. Please consult RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
receiptcre	Receipt Create Message	ReceiptDesc.dtd
receiptmod	Receipt Modify (Adjustment) Message	ReceiptDesc.dtd

Design Assumptions

1. The stock order subscription process supports the break-to-sell functionality. Transfers, allocations and shipments in RMS will only contain break to sell orderable items. Inventory adjustment and stock ledger will be performed on the orderable only, not the sellable.
2. The stock order and order subscription process supports the catch weight functionality. It is assumed that a break-to-sell sellable item cannot be a simple pack catch weight item.
3. Catch weight functionality is not completely rounded out in this release. For instance, it is **not** applied to the following areas:
   a. Any of the retail calculations (including total_retail on TRAN_DATA and retail markup/markdown);
   b. The total amount on SUP_DATA;
   c. Open to buy buckets;
   d. When a catch weight component item's standard UOM is a MASS UOM, TRAN_DATA.units is based on V_PACKSKU_QTY.qty instead of the actual weight.
4. An externally generated transfer will contain physical locations. When system options INTERCOMPANY_TSF_IND = 'Y', the stock order receiving process currently does **not** support the receiving of an externally generated transfer that involves a warehouse to warehouse transfer. This is because a physical location does **not** have transfer entities.
5. PO receiving does not handle break to sell items.

Tables

TABLE	SELECT	INSERT	UPDATE	DELETE
TSFHEAD	Yes	No	Yes	No
TSFDETAIL	Yes	Yes	Yes	No
ALLOC_HEADER	Yes	No	Yes	No
ALLOC_DETAIL	Yes	No	Yes	No
ORDHEAD	Yes	No	Yes	No
ORDSKU	Yes	Yes	Yes	No
ORDLOC	Yes	Yes	Yes	No
SHIPMENT	Yes	Yes	Yes	No
SHIPSKU	Yes	Yes	Yes	No
TRAN_DATA	No	Yes	No	No
SUP_DATA	No	Yes	No	No
ITEM_LOC_SOH	Yes	Yes	Yes	No
ITEM_LOC	Yes	Yes	No	No
ITEM_ZONE_PRICE	Yes	Yes	No	No
PRICE_HIST	No	Yes	No	No
SHIPITEM_INV_FLOW	Yes	Yes	Yes	No
MRT_ITEM_LOC	Yes	No	Yes	No
APPT_DETAIL	Yes	No	Yes	No
DOC_CLOSE_QUEUE	No	Yes	No	No
DUMMY_CARTON_STAGE	No	Yes	No	No
ALC_HEAD	Yes	Yes	Yes	No
CONTRACT_HEADER	Yes	No	Yes	No
CONTRACT_DETAIL	Yes	No	Yes	No
INVC_MATCH_WKSHT	Yes	No	Yes	No
INVC_HEAD	Yes	Yes	Yes	No
INVC_DETAIL	Yes	Yes	Yes	No
INVC_TOLERANCE	Yes	Yes	Yes	Yes
INVC_XREF	Yes	Yes	No	No
INVC_MATCH_VAT	Yes	Yes	Yes	No
TERMS	Yes	No	No	No
SUPS	Yes	No	No	No

TABLE	SELECT	INSERT	UPDATE	DELETE
VAT_REGION	Yes	No	No	No
DEPS	Yes	No	No	No
WEEK_DATA	Yes	No	No	No
MONTH_DATA	Yes	No	No	No
ITEM_SUPP_COUNTRY	Yes	No	No	No
ITEM_SUPP_COUNTRY_LOC	Yes	Yes	No	No
ITEM_SUPP_COUNTRY_DIM	Yes	No	No	No
UOM_CLASS	Yes	No	No	No
NWP	Yes	Yes	Yes	No
STORE	Yes	No	No	No
WH	Yes	No	No	No
ITEM_MASTER	Yes	No	No	No
V_PACKSKU_QTY	Yes	No	No	No
ITEM_XFORM_HEAD	Yes	No	No	No
ITEM_XFORM_DETAIL	Yes	No	No	No
CURRENCIES	Yes	No	No	No
CURRENCY_RATES	Yes	No	No	No
PERIOD	Yes	No	No	No
SYSTEM_OPTIONS	Yes	No	No	No

Regular Price Change

Functional Area

RMS subscribing to regular price changes

Business Overview

RPM sends approved price changes for regular price items to external subsystems. Additionally, RMS needs to know this information so it can set the effective date of the price change on the TICKET_REQUEST table.

A subscription family will be created to bring price changes from RPM to RMS. The new sub family will be called RegPrcChg. RMS will subscribe the create, mod and delete of price change messages.

This API creates, modifies or deletes price change records on the TICKET_REQUEST table. Packs and warehouse locations are ignored.

Once the price changes have already been downloaded by a batch program, there could be instances where a new price change is updated or deleted for the same price change ID/item/loc/date combination. In this case, the latest instance of price change is inserted or updated in TICKET_REQUEST table.

Package Impact

Filename: rmssub_regprcchgtkts/b.pls

```
CONSUME(O_status_code     OUT       VARCHAR2,
        O_error_message   OUT       RTK_ERRORS.RTK_TEXT%TYPE,
        I_message         IN        RIB_OBJECT,
        I_message_type    IN        VARCHAR2);
```

This procedure initially checks that the passed in message type is a valid type for regular price change messages. The valid message types are: regprcchgCre, regprcchgMod and regprcchgDel. If the message type is invalid, a status of “E” should be returned to the external system along with an appropriate error message informing the external system that the status is invalid.

If the message type is valid, the generic RIB_OBJECT will need to be downcast to the actual object using the Oracle’s treat function. There will be an object type that corresponds with each message type. If the downcast fails, a status of “E” should be returned to the external system along with an appropriate error message informing the external system that the object passed in is invalid.

If the downcast is successful, then consume needs to verify that the message passes all of the RMS business validation. It does not actually perform any validation itself; instead, it calls the RMSSUB_REGPRCCHG_TKT_VALIDATE.CHECK_MESSAGE function to determine whether the message is valid. This function is overloaded so simply passing the object in should be sufficient. If the message passed RMS business validation, then the function will return true; otherwise, it will return false. If the message has failed RMS business validation, a status of “E” should be returned to the external system along with the error message returned from the CHECK_MESSAGE function.

Once the message has passed RMS business validation, it can be persisted to the RMS database. The consume function does not have to have any knowledge of how to persist the message to the database, it calls the RMSSUB_REGPRCCHG_TKT_SQL.PERSIST() function. This function is overloaded, so simply passing the object should be sufficient. If the database persistence fails, the function will return false. A status of “E” should be returned to the external system along with the error message returned from the PERSIST() function.

Once the message has been successfully persisted, there is nothing more for the consume procedure to do. A success status, “S”, should be returned to the external system indicating that the message has been successfully received and persisted to the RMS database.

RMSSUB_REGPRCCHG_TKT.HANDLE_ERROR() – This is the standard error handling function that wraps the API_LIBRARY.HANDLE_ERROR function.

Business Validation Module

File name: rmssub_regprcchg_tktvals.pls

```
Global Function: CHECK_MESSAGE
               (O_error_mesage       OUT     RTK_ERRORS.RTK_TEXT%TYPE,
                O_ticket_tbl         OUT     TICKET_SQL.TICKET_TBL,
                I_message            IN      RIB_REGPRCCHGDESC_REC,
                I_message_type       IN      VARCHAR2)
```

This overloaded function performs all business validation associated with ticket request create, modify and delete messages. It is important that the signature uses IN for the message and not IN OUT. When IN is used, the parameter is passed by reference. Passing by reference keeps the server from duplicating the memory allocation.

Call the CHECK_REQUIRED_FIELDS function to make sure all required fields are not NULL. Call the CHECK_EXISTENCE function to see if a record for the item/loc/price change ID already exists on the TICKET_REQUEST table. If it exists and the message type is create, treat it as a Mod message type. If it does **not** exist and the message type is Mod, treat it as a Create message type. If it does not exist and the message type is Delete, return TRUE since no further processing is needed. Call VALIDATE_MESSAGE function to check for validity of the required fields.

Finally, the ticket record used for DML is populated within the POPULATE_RECORD function and passed back to RMSSUB_REGPRCCHG_TKT.CONSUME.

Internal Functions

CHECK_REQUIRED_FIELDS

This overloaded function ensures that all required fields in the message are NOT NULL.

VALIDATE_MESSAGE

This overloaded function validates the values of the message. It will also populate the fields on TICKET_REQUEST that were not included in the message.

- If the location type is warehouse or the item is a pack, no further processing is needed.
- Stores are validated along with the currency code.
- Validate that the item is at the transactional level.

POPULATE_RECORD

This overloaded function populates the ticket request output record with the values sent in the message.

Bulk or single DML module

All insert, update and delete SQL statements are located in the family packages. The private functions call these packages.

File name: rmssub_regprcchg_tktsqls.pls

```
PERSIST
       (O_error_mesage       OUT          RTK_ERRORS.RTK_TEXT%TYPE,
        I_message            IN           TICKET_SQL.TICKET_TBL,
        I_upd_message        IN           TICKET_SQL.TICKET_TBL,
        I_message_type       IN           VARCHAR2)
```

This function will perform INSERT/UPDATE/DELETE statements by calling the appropriate functions according to the message type and passing the data in a record to these functions.

For the message type indicating a ticket request insert or update, call the TICKET_SQL.NEW_PRICE_CHANGE and TICKET_SQL.UPDATE_PRICE_CHANGE functions with the ticket record. For the message type indicating a delete, call TICKET_SQL.DELETE_PRICE_CHANGE function to delete the record from TICKET_REQUEST.

File name: tickets/b.pls

```
NEW_PRICE_CHANGE
          (O_error_message          OUT          RTK_ERRORS.RTK_TEXT%TYPE,
           I_ticket_tbl             IN           TICKET_SQL.TICKET_TBL)
```

This function inserts records into the TICKET_REQUEST table.

```
UPDATE_PRICE_CHANGE
          (O_error_message          OUT          RTK_ERRORS.RTK_TEXT%TYPE,
           I_ticket_tbl             IN           TICKET_SQL.TICKET_TBL)
```

This function calls LOCK_TICKET function to lock and update the TICKET_REQUEST table.

```
LOCK_TICKET (O_error_message        OUT      RTK_ERRORS.RTK_TEXT%TYPE,
             I_ticket_tbl           IN       TICKET_SQL. TICKET_TBL)
```

This function locks the records on TICKET_REQUEST table.

```
DELETE_PRICE_CHANGE
               (O_error_message      OUT     RTK_ERRORS.RTK_TEXT%TYPE,
                I_ticket_tbl         IN      TICKET_SQL.TICKET_TBL)
```

This function calls LOCK_TICKET function to lock and delete records from the TICKET_REQUEST table.

Message DTD

Here are the filenames that correspond with each message type. Please consult the RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
RegPrcChgCre	Regular Price Change Create Message	RegPrcChgDesc.dtd
RegPrcChgMod	Regular Price Change Modify Message	RegPrcChgDesc.dtd
RegPrcChgDel	Regular Price Change Delete Message	RegPrcChgDesc.dtd

Design Assumptions

- One of the primary assumptions in the current API approach is that ease of code will outweigh performance considerations. It is hoped that the 'trickle' nature of the flow of data will decrease the need to dwell on performance issues and instead allow developers to code in the easiest and most straight forward manner.
- The adaptor is only setup to call stored procedures, not stored functions. Any public program then needs to be a procedure.

Tables

TABLE	SELECT	INSERT	UPDATE	DELETE
TICKET_REQUEST	Yes	Yes	Yes	Yes
ITEM_LOC_SOH	Yes	No	No	No
STORE	Yes	No	No	No
ITEM_LOC	Yes	No	No	No
ITEM_TICKET	Yes	No	No	No
ITEM_MASTER	Yes	No	No	No

Return to Vendor (RTV)

Functional Area

RTV subscription

Business Overview

RMS subscribes to return-to-vendor (RTV) messages from the RIB. When an RTV is shipped out from a warehouse or store, the RTV information is sent from the external system (such as RWMS) to the RIB. RMS subscribes to the RTV information as published from the RIB and places the information onto RMS tables, depending on the validity of the records enclosed within the message.

The RTV message can be processed as a flat message when the header description contains information for one RTV item. The message can also be processed as a hierarchical message when the detail node is populated with one or more RTV items. RMS primarily uses these messages to update inventory quantities and stock ledger values.

Subscription Package

Filename: rmssub_rtvs/b.pls

```
RMSSUB_RTV.CONSUME
               (O_status_code        IN OUT        VARCHAR2,
                O_error_message      IN OUT        VARCHAR2,
                I_message            IN            RIB_OBJECT,
                I_message_type       IN            VARCHAR2)
```

This procedure will need to initially ensure that the passed in message type is a valid type for RTV messages. The valid message types for RTV messages are listed in the Message DTD section below.

If the message type is invalid, a status of “E” would be returned to the external system along with an appropriate error message informing the external system that the message type is invalid. If the message type is valid, the generic RIB_OBJECT will be downcast to the actual object using the Oracle’s treat function. If the downcast fails, a status of “E” is returned to the external system along with an appropriate error message informing the external system that the object passed in is invalid.

If the downcast is successful, then consume will call PARSE_RTV to parse the RTV message and PROCESS_RTV to perform business validation and desired functionality. Any time the message fails business validation, a status of “E” is returned to the external system along with an appropriate error message.

Once the message has been successfully processed, a success status, “S”, is returned to the external system indicating that the message has been successfully received and persisted to the RMS database.

PARSE_RTV

This function parses the RIB_OBJECT and builds an API rtv_record for processing.

PROCESS_RTV

This function calls RTV_SQL.APPLY_PROCESS to perform all business validation and desired functionality associated with a RTV message.

For break to sell items, if a sellable only item is on the message, call CHECK_ITEMS and GET_ORDERABLE_ITEMS to convert the sellable item(s) to the corresponding orderable item(s). The orderable items will be inserted or updated on the tables affected by an RTV.

The RTV_SQL.APPLY_PROCESS is called for each of the orderable items and each of the regular items.

CHECK_ITEMS

This function separates the item details on the message into two groups: one contains sellable only items and one contains regular items.

GET_ORDERABLE_ITEMS

This function builds a collection of orderable items based on the sellable items. It calls ITEM_XFORM_SQL.RTV_ORDERABLE_ITEM_INFO to distribute the sellable quantities among the orderable items.

Filename: rtvs/b.pls

RTV_SQL.APPLY_PROCESS

This function performs business validation and desired functionality for a RTV message. It includes the following:

- Verify that an orderable but non-sellable and non-inventory item **cannot** be an RTV item.
- Verify that an RTV item must be a tran-level or above tran-level item.
- If the RTV item is a simple pack catch weight item, verify that weight and weight unit of measure (UOM) are either both defined or both NULL, and weight UOM is in the MASS UOM class.
- Verify that the item supplier relation exists.
- Verify that the location is a valid store or warehouse.
- Verify that the item/loc relation exists.
- If returning a pack to a warehouse, the pack must be received as pack at the warehouse.
- Verify that from disposition is a valid inventory status code (on INV_STATUS_CODES).
- Verify that the reason code is a valid RTV reason code (code type ‘RTVR’ on CODE_DETAIL).
- For an externally generated RTV, if multi-channel is on and the location is a warehouse, then physical location is on the message. RTV quantity will be distributed among the virtual locations of the physical location.
- Check the existence of RTV in RTV_HEAD based on: a) rtv_order_no; b) ext_ref_no and location. An RTV will be updated if it already exists and inserted if not. The RTV will be marked as shipped.
- Check the existence of RTV item in RTV_DETAIL based on: rtv_order_no, item, reason and inventory status. An RTV_DETAIL will be updated if it already exists and inserted if not.

- If the RTV item is a content item of a deposit item, RTV_DETAIL will be inserted or updated for the associated container item.
- Determine RTV unit cost as the following:
  - Use the unit cost on the RTV message if defined. It is in location currency. Otherwise,
  - Use RTV_DETAIL.unit_cost if exists. It is in supplier currency. Otherwise,
  - Use the last receipt cost if exists. It is in location currency. Otherwise,
  - Use item's WAC at the location. It is in location currency.
    The unit cost is used to evaluate the cost of the RTV goods. The cost values on RTV tables are written in supplier currency, but all TRAN_DATA records are written in location currency.
- If the RTV item is a simple pack catch weight item, the total RTV cost is based on weight.
- Update the following stock buckets on ITEM_LOC_SOH: RTV_QTY, STOCK_ON_HAND, PACK_COMP_SOH. For a simple pack catch weight item at the warehouse, also update average weight.
- Write the following TRAN_DATA records:
  - 24 – for RTV. It writes units, total_cost and total_retail.
  - 71/72 – for cost variance between item's WAC at the location and RTV unit cost. It writes units and total_cost.
  - 65 – for restocking fees. For a non-MRT type of RTV, the restocking fee is written for the RTV location. For an MRT type of RTV, the restocking fee is distributed among the MRT locations. It writes units and total_cost.
  - 22 – for stock adjustment, if stock counting has already happened at the store for the item.

  If the RTV item is a pack, TRAN_DATA is written for component items. If the RTV location is a physical warehouse, TRAN_DATA is written for virtual locations. TRAN_DATA total cost and total retail are always written in location currency.
- If system options ext_invc_match_ind is on, create or update INVC_HEAD and INVC_DETAIL for the RTV.

Message DTD

Here are the filenames that correspond with each message type. Please consult the RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
rtvcre	RTV Create Message	RTVDesc.dtd

Design Assumptions

1. Catch weight functionality is not completely rounded out in this release. For instance, it is **not** applied to the following areas:
   - Any of the retail calculations (including total_retail on TRAN_DATA and retail markup/markdown);
   - The total amount on SUP_DATA;
   - Open to buy buckets;
   - When a catch weight component item's standard UOM is a MASS UOM, TRAN_DATA.units is based on V_PACKSKU_QTY.qty instead of the actual weight.
2. MRT RTV can only be created in RMS. Therefore it will only contain virtual locations. Physical location distribution logic does **not** apply to MRT RTVs.

Tables

TABLE	SELECT	INSERT	UPDATE	DELETE
RTV_HEAD	Yes	Yes	Yes	No
RTV_DETAIL	Yes	Yes	Yes	No
ITEM_LOC_SOH	Yes	No	Yes	No
TRAN_DATA	No	Yes	No	No
INV_STATUS_CODES	Yes	No	No	No
CODE_DETAIL	Yes	No	No	No
ITEM_MASTER	Yes	No	No	No
ITEM_SUPPLIER	Yes	No	No	No
ITEM_SUPP_COUNTRY	Yes	No	No	No
ITEM_LOC	Yes	No	No	No
STORE	Yes	No	No	No
WH	Yes	No	No	No
SHIPMENT	Yes	No	No	No
SHIPSKU	Yes	No	No	No
DEPS	Yes	No	No	No
SUPS	Yes	No	No	No
ADDR	Yes	No	No	No
UOM_CLASS	Yes	No	No	No
V_PACKSKU_QTY	Yes	No	No	No
MRT_ITEM_LOC	Yes	No	No	No
ITEM_XFORM_HEAD	Yes	No	No	No

TABLE	SELECT	INSERT	UPDATE	DELETE
ITEM_XFORM_DETAIL	Yes	No	No	No
INVC_HEAD	Yes	Yes	Yes	Yes
INVC_DETAIL	Yes	Yes	No	Yes
INVC_NON_MERCH	No	Yes	No	Yes
INVC_MERCH_VAT	Yes	Yes	Yes	Yes
INVC_DETAIL_VAT	Yes	No	No	Yes
INVC_MATCH_QUEUE	Yes	No	No	Yes
INVC_DISCOUNT	Yes	No	No	Yes
INVC_TOLERANCE	Yes	No	No	Yes
ORDLOC_INVC_COST	Yes	No	Yes	No
NON_MERCH_CODE_HEAD	Yes	No	No	No
SYSTEM_OPTIONS	Yes	No	No	No

Stock Order (SO) Status

Functional Area

Stock Order Status.

Business Overview

RMS subscribes to stock order status messages from the RIB. Stock order status messages are published by an external application, such as a warehouse management system. (RWMS, for example). RMS uses the data contained in the messages to:

- Update the following tables when the status of the 'distro' changes at the warehouse:
  - ALLOC_DETAIL
  - ITEM_LOC_SOH
  - TSF_DETAIL
- Determine when the warehouse is processing a transfer or allocation. In-process transfers or allocations cannot be edited and are determined by the initial and final quantities to be filled by the external system.

Stock Order Status Explanations

The following tables describe the stock order statuses for both transfers and allocation document types and what occurs in RMS after receiving the respective status.

Stock order status received in message on a TRANSFER (where 'distro_document_type' = 'T')	What RMS does
DS (Details Selected) When RWMS publishes a message on a transfer with a status of DS (Details Selected), RMS will increase the selected quantity on TSFDETAIL for the transfer/item combination.	Increase tsfdetail.selected_qty
DU (Details Un-selected) When RWMS publishes a message on a transfer with a status of DU (Details Un-Selected), RMS decreases the selected quantity on TSFDETAIL for the transfer/item combination.	Decrease tsfdetail.selected_qty

Stock order status received in message on a TRANSFER (where 'distro_document_type' = 'T')	What RMS does
NI (WMS Line Cancellation) When RWMS publishes a message on a transfer with a status of NI (No Inventory – WMS Line Cancellation), RMS will decrease the selected quantity by the quantity on the message. RMS will also increase the cancelled quantity, decrease the transfer quantity, decrease the reserved quantity* for the from location, and decrease the expected quantity* for the to location by the lesser of 1.) the quantity on the message; 2.) the transfer quantity – shipped quantity. *If the transfer status is not Closed.	Decrease tsfdetail.select_qty, and tsfdetail.tsf_qty increase tsfdetail.cancelled_qty, decrease item_loc_soh.tsf_reserved_qty for the from location and item_loc_soh.tsf_expected_qty for the from location
PP (Distributed) When RWMS publishes a message on a transfer with a status of PP (Pending Pick - Distributed), RMS will decrease the selected quantity and increase the distro quantity.	Decrease tsfdetail.selected_qty, increase tsfdetail.distro_qty
PU (Un-Distribute) When RWMS publishes a message on a transfer with a status of PU (Un-Distribute), RMS will decrease the distributed qty.	Decrease tsfdetail.distro_qty
RS (Return To Stock) When RWMS published a message on a transfer with a status of RS (Return To Stock), RMS will decrease the distributed qty.	Decrease tsfdetail.distro_qty
EX (Expired) When RWMS publishes a message on a transfer with a status of EX (Expired), RMS will increase the cancelled quantity, decrease the transfer quantity, decrease the reserved quantity* for the from location, and decrease the expected quantity* for the to location by the lesser of 1.) the quantity on the message; 2.) the transfer quantity – shipped quantity. *If the transfer status is not Closed.*	Increase tsfdetail.cancelled_qty, decrease tsfdetail.tsf_qty, item_loc_soh.tsf_reserved_qty for the from location and item_loc_soh.tsf_expected_qty for the to location

Stock order status received in message on a TRANSFER (where 'distro_document_type' = 'T')	What RMS does
SR (Store Reassign) When RWMS publishes a message on a transfer with a status of SR (Store Reassign) the quantity can be either positive or negative. In either case it will be added to the distro_qty (adding a negative will have the same affect as subtracting it). If it is positive RMS will also decrease the selected_qty.	Add to tsfdetail.distro_qty, decrease tsfdetail.selected_qty (when the input qty < 0)

Stock order status received in message on an ALLOCATION (where 'distro_document_type' = 'A')	What RMS does
DS (Details Selected) When RWMS publishes a message on an allocation with a status of DS (Details Selected), RMS will increase the selected quantity on alloc_detail for the allocation/item/location combination.	Increase alloc_detail.selected_qty
DU (Details Un-Selected) When RWMS publishes a message on an allocation with a status of DU (Details Un-Selected), RMS will decrease the selected quantity on alloc_detail for the allocation/item combination.	Decrease alloc_detail.selected_qty
NI (WMS Line Cancellation) When RWMS publishes a message on an allocation with a status of NI (No Inventory – WMS Line Cancellation), RMS will decrease the selected quantity by the quantity on the message. RMS will also increase the cancelled quantity, decrease the allocated quantity, decrease the reserved quantity* for the from location, and decrease the expected quantity* for the to location by the lesser of 1.) the quantity on the message; 2.) the transfer quantity – shipped quantity. **If the allocation status is not Closed and the allocation is a stand alone allocation.*	Decrease alloc_detail.qty_ selected and alloc_detail.qty_allocated, increase alloc_detail.cancelled_qty, decrease item_loc_soh.tsf_reserved_qty for the from location and item_loc_soh.tsf_expected_qty for the to location

Stock order status received in message on an ALLOCATION (where 'distro_document_type' = 'A')	What RMS does
PP (Distributed) When RWMS publishes a message on an allocation with a status of PP (Pending Pick - Distributed), RMS will decrement the selected quantity and increment the distro quantity.	Decrease alloc_detail.qty_selecteded, increase alloc_detail.qty_distro
PU (Un-Distribute) When RWMS publishes a message on an allocation with a status of PU (Un-Distribute), RMS will decrease the distributed qty.	Decrease alloc_detail.qty_distro
RS (Return to Stock) When RWMS published a message on an allocation with a status of RS (Return to Stock), RMS will decrease the distributed qty.	Decrease alloc_detail.qty_distro
EX (Expired) When RWMS publishes a message on an allocation with a status of EX (Expired), RMS will increase the cancelled quantity, decrease the allocated quantity, decrease the reserved quantity* for the from location, and decrease the expected quantity* for the to location by the lesser of 1.) the quantity on the message; 2.) the transfer quantity – shipped quantity. **If the allocation status is not Closed and the allocation is a stand alone allocation.*	Decrease alloc_detail.qty_allocated, increase alloc_detail.qty_cancelled, decrease item_loc_soh.tsf_reserved_qty for the from location and item_loc_soh.tsf_expected_qty for the to location
SR (Store Reassign) When RWMS publishes a message on an allocation with a status of SR (Store Reassign) the quantity can be either positive or negative. In either case, it will be added to the qty_distro (adding a negative will have the same affect as subtracting it). If it is positive, RMS will also decrease the qty_selected.	Add to alloc_detail.qty_distro, decrease alloc_detail.qty_selected (when the input qty < 0)

Pack Considerations

Whenever the from location is a warehouse, check if the item is a pack or an each. If the item is not a pack item, no special considerations are necessary. For each warehouse-pack item, check the receive_as_type on ITEM_LOC to determine if it is received into the warehouse as a pack or a comp item. If it is received as an each, update ITEM_LOC_SOH for the comp item. If it is received as a pack, update ITEM_LOC_SOH for the pack item and the comp item.

A stock order is an outbound merchandise request from a warehouse or store. In RMS, a stock order takes the form of either a transfer or allocation.

Stock order status upload receives a message from the RIB, published from RWMS, communicating the status of a specific stock order. This communication provides for the synchronization of data between RWMS and RMS. The information from RWMS has only one level, in other words no detail records. This information is used to update the TSFDETAIL, ALLOC_DETAIL and ITEM_LOC_SOH tables.

Package Impact

Filename: rmssub_sostatuss/b.pls

Public API Procedures

```
RMSSUB_SOSTATUS.CONSUME
               (O_status_code        IN OUT        VARCHAR2,
                O_error_message      IN OUT        VARCHAR2,
                I_message            IN            RIB_SOStatusDesc_REC,
                I_message_type       IN            VARCHAR2);
```

This procedure accepts Stock Order Status information in the form of an Oracle Object data type from the RIB (I_message) and a message type of ‘sostatuscre’. The procedure will first call the RESET function to initialize internal variables. The procedure will then extract the values from the oracle object. These will then be passed on to private internal functions which will validate the values and place them on the database depending upon the success of the validation.

```
VALIDAT
               (O_error_message      IN OUT        VARCHAR2,
                O_exist              IN OUT        BOOLEAN,
                I_distro_number      IN            VARCHAR2);
```

Validation:

Validate the distro is valid. A distro refers to either a transfer or an allocation.

Internal Functions and Procedures (rmssub_frttermcre.pls):

Function HANDLE_ERRORS():

```
HANDLE_ERRORS
               (O_status             IN OUT        VARCHAR2,
                IO_error_message     IN OUT        VARCHAR2,
                I_cause              IN            VARCHAR2,
                I_program            IN            VARCHAR2);
```

Error Handling

If an error occurs in this procedure or any of the internal functions, this procedure places a call to HANDLE_ERRORS in order to parse a complete error message and pass back a status to the RIB.

This function is used to put error handling in one place in order to make future error handling enhancements easier to implement. The function should consist of a call to API_LIBRARY.HANDLE_ERRORS. API_LIBRARY.HANDLE_ERRORS accepts a program name, the cause of the error and potentially an unparsed error message if one has been created through a call to SQL_LIB.CREATE_MESSAGE. The function uses these input variables to parse a complete error message and pass back a status, depending upon the message and error type, back up through the consume function and up to the RIB.

PARSE_SOS

This function first calls VALIDATE to check that the transfer or allocation from the oracle object exists in RMS. If the transfer or allocation exists, the function breaks down the message into its component parts and sends these parts into PROCESS_SOS.

PROCESS_SOS

Based on the status sent from RWMS, quantity fields on either TSFDETAIL or ALLOC_DETAIL and ITEM_LOC_SOH are updated.

A new ‘status’ was created that is used exclusively by stores to cancel requested quantities that have not yet been shipped. The status is ‘SC’, for Store Cancellation.

UPDATE_TSF

Updates the record on TSF_DETAIL if the message is for a transfer.

UPDATE_ALLOC

Updates the record on ALLOC_DETAIL if the message is for an allocation.

UPD_FROM_ITEM_LOC

Updates item_loc_soh.tsf_reserved_qty for the from location. If the comp_level_upd indacator is ‘Y’ then it will also update the item_loc_soh.pack_comp_resv field for the item passed in.

UPD_TO_ITEM_LOC

Updates item_loc_soh.tsf_expected_qty for the to location. If the comp_level_upd indacator is ‘Y’ then it will also update the item_loc_soh.pack_comp_exp field for the item passed in.

GET_RECEIVE_AS_TYPE

This function gets the receive as type value from ITEM_LOC for the passed-in item and location combination.

POPULATE_DOC_CLOSE_QUEUE

This function is called to populate an array which holds stock order information that will be placed on the DOC_CLOSE_QUEUE table.

RESET

This function deletes any values that are currently held in the package's global variables.

DO_BULK

This function is used to do bulk inserts or updates of the ALLOC_DETAIL, TSFDETAIL, TSFHEAD and DOC_CLOSE_QUEUE tables. The tables are updated/inserted using the arrays that were built in the rest of the package.

Message DTD

Here are the filenames that correspond with each message type. Please consult the RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
sostatuscre	Stock Order Status Create Message	SOStatusDesc.dtd

Design Assumptions

- One of the primary assumptions in the current API approach is that ease of code will outweigh performance considerations. It is hoped that the 'trickle' nature of the flow of data will decrease the need to dwell on performance issues and instead allow developers to code in the easiest and most straight forward manner.
- The adaptor is only setup to call stored procedures, not stored functions. Any public program then needs to be a procedure.

Tables

TABLE	SELECT	INSERT	UPDATE	DELETE
ITEM_LOC_SOH	Yes	Yes	Yes	No
ITEM_LOC	Yes	No	No	No
ALLOC_DETAIL	Yes		Yes	No
ALLOC_HEADER	Yes	No	No	No
TSFDETAIL	Yes		Yes	No
TSFHEAD	Yes	No	Yes	No
DOC_CLOSE_QUEUE	No	Yes	No	No
V_PACKSKU_QTY	Yes	No	No	No

Stock Count Schedule (SCH)

Functional Area

Inventory – Stock Counts

Business Overview

Stock count schedule messages are published to the RIB by an integration subsystem to communicate unit and value stock count schedules to RMS. RMS uses stock count schedule data to help synchronize the inventories of another application and RMS. The other application performs a physical inventory count and uploads the results, and RMS compares the discrepancies.

This API allows external systems to create, update, and delete stock counts within RMS. Only Unit and Dollar stock counts (stocktake_type = ‘B’) are subscribed by RMS at this time. Department, class and subclass can be null; if not provided a full count is presumed.

If the other application requires at year-end to consolidate annual and booking numbers, the annual count can be initiated by the other application and uploaded into RMS. RMS accepts the unit variances and processes these automatically. The dollar values will need user input from the central office.

Consume Module

Filename: rmssub_stakeschedules/b.pls

```
PROCEDURE CONSUME
               (O_status_code         IN OUT         VARCHAR2,
                O_error_message       IN OUT         VARCHAR2,
                I_message             IN             RIB_OBJECT,
                I_message_type        IN             VARCHAR2);
```

Package to subscribe to stock count schedule message, parse the details, and pass them into a new stock schedule package.

- If the message type is StkCountSchDel, validate before deleting the cycle count.
- For other message types, business validations are performed before creating or updating the cycle count.
- Once the message has been successfully processed, there is nothing more for the consume procedure to do. A success status, “S”, should be returned to the external system indicating that the message has been successfully received and persisted to the RMS database.

Business Validation and DML Module

Filename: stake_schedules/b.pls

Package to validate stock schedule data and insert/update to the stock count tables.

VALIDATE_VALUES

- Cannot delete a cycle count if it has been processed.
- Cannot update a cycle count that has started or has been set to be deleted.
- Cannot process anything if stock count is currently locked

VALIDATE_HIERARCHY

- Unit and Dollar stock counts at a warehouse must be at the department level only.
- Validate department, class and subclass.

VALIDATE_LOCATION

- Only stockholding (virtual) warehouses can be on a stock count.

PROCESS_PROD

- Validate and create a STAKE_PRODUCT record. No validation is done if the record is passed in for initial processing.

PROCESS_LOC

- Validate and create a STAKE_LOCATION record. No validation is done if the record is passed in for initial processing.

PROCESS_DEL

CREATE_SH_REC

- Create a record for STAKE_HEAD.

CREATE_SP_REC

- Create a STAKE_PRODUCT record.

DELETE_RECS

- Delete from STAKE_PRODUCT and STAKE_LOCATION tables.

Message DTD

Here are the filenames that correspond with each message type. Please consult RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
StkCountSchCre	Stock Count SCH Create Message	StkCountSchDesc.dtd
StkCountSchMod	Stock Count SCH Modify Message	StkCountSchDesc.dtd
StkCountSchDel	Stock Count SCH Delete Message	StkCountSchRef.dtd

Design Assumptions

- The store is locked down during the day of the stock count. This means no receipts or sales during the 24 hour period of that day.
- After the stock counts have been uploaded into RMS no correction will be possible. Validation of the accurate value of the stock count is the responsibility of the on site-auditing manager during the stock count process. Corrections will have to be made through a unit count or inventory adjustment.
- Due to large volume the stock count results are interfaced in the form of a flat file.
- RMS assumes that the interfaced item information is complete for that merchandise hierarchy that is counted. The RMS system attempts to auto process this information. The other application has ownership of the inventory, and it is self evident that when RMS receives the information it is complete.

Tables

TABLE	SELECT	INSERT	UPDATE	DELETE
DEPS	Yes	No	No	No
STORE	Yes	No	No	No
WH	Yes	No	No	No
STAKE_HEAD	Yes	Yes	Yes	No
STAKE_PRODUCT	No	Yes	No	Yes
STAKE_LOCATION	No	Yes	No	Yes
SYSTEM_OPTIONS	Yes	No	No	No

Store

Functional Area

Store

Design Overview

The RMS system option 'sor_org_hier_ind' indicates whether RMS is the system of record for organizational hierarchy information.

When RMS is **not** the system of record, the Store Subscription API provides the ability to keep store data in RMS in sync with an external system by using data from that system to create, edit, and delete stores within RMS. The store data handled by the API includes basic store data plus relationship data between stores and their location traits and walk-through stores. This data can be viewed but not updated in RMS.

Basic Store Data

When creating a new store in RMS, the API uses RMS store creation batch logic. When a store creation message is received, it is validated and placed onto a staging table. The store creation batch program in RMS reads from this table and actually creates the store in RMS. This is done to reuse existing validation logic within the store creation program. One consequence of this approach is that the API is not able to communicate whether the store was successfully created in RMS, only that the message was received.

When updating an existing store in RMS, the API performs the update immediately upon message receipt, so success or failure can be communicated to the calling application.

The API will also handle store delete messages. Like the store creation message subscription process, stores will not actually be deleted from the system upon receipt of the message. However, validation will occur to ensure the store can be deleted. The store will be added to the DAILY_PURGE table. When the daily purge batch process runs, the store will be removed from the system. Because the store is not deleted when the message is received, the API is not able to communicate whether the removal of the store from RMS has been successful, only that the message was successfully received.

Location Trait and Walk-Through Store Data

By default, stores inherit the location traits of the district to which they belong. However, specific location traits can also be assigned at the store level. Using the incoming external data, the API will create or delete relationships between stores and existing location traits, or between stores and walk-through stores. (Note: This API does not create or delete location traits; that is handled by the Location Traits subscription.)

Location trait and walkthrough store data cannot be sent in on a store create message. The store create program must first process the store before it can have details attached to it.

Location trait and walkthrough store data must be processed separately as they each have their own distinct message types. These detail create messages will contain a snapshot of the store record. Note that location traits must already exist prior to being added to the store.

Deletion of location trait and walkthrough store relationships will also be handled within this API. The detail delete messages must be processed separately as they each have their own distinct message types.

Consume Module

Rmssub_xstores/b.pls

```
RMSSUB_XSTORE.CONSUME
               (O_status_code        IN OUT        VARCHAR2,
                O_error_message      IN OUT        VARCHAR2,
                I_message            IN            RIB_OBJECT,
                I_message_type       IN            VARCHAR2)
```

This procedure will need to initially ensure that the passed in message type is a valid type for store messages. If the message type is invalid, a status of 'E' should be returned to the external system along with an appropriate error message informing the external system that the status is invalid.

If the message type is valid, the generic RIB_OBJECT will need to be downcast to the actual object using the Oracle's treat function. If the downcast fails, a status of 'E' should be returned to the external system along with an appropriate error message informing the external system that the object passed in is invalid.

If the downcast is successful, then consume will need to verify that the message passes all of the RMS business validation. It does not actually perform any validation itself, instead, it will call the RMSSUB_XSTORE_VALIDATE.CHECK_MESSAGE function to determine whether the message is valid. If the message has failed RMS business validation, a status of 'E' should be returned to the external system along with the error message returned from the CHECK_MESSAGE function.

Once the message has passed RMS business validation, it can be persisted to the RMS database. The consume function will call the RMSSUB_XSTORE_SQL.PERSIST_MESSAGE() function. If the database persistence fails, the function will return false. A status of 'E' should be returned to the external system along with the error message returned from the PERSIST_MESSAGE() function.

Once the message has been successfully persisted, there is nothing more for the consume procedure to do. A success status, 'S', should be returned to the external system indicating that the message has been successfully received and persisted to the RMS database.

RMSSUB_XSTORE.HANDLE_ERROR() – This is the standard error handling function that wraps the API_LIBRARY.HANDLE_ERROR function.

Business Validation Module

It should be noted that some of the business validation is referential or involves uniqueness. This validation is handled automatically by the referential integrity constraints and the unique indexes implemented on the database.

Filename: rmssub_xstorevals/b.pls

```
RMSSUB_XSTORE_VALIDATE.CHECK_MESSAGE
           (O_error_message   IN OUT         VARCHAR2,
            O_store_rec       OUT            NOCOPY  STORE_SQL.STORE_ROW_TYPE,
            I_message         IN             RIB_XStoreDesc,
            I_message_type    IN             VARCHAR2)
```

This function performs all business validation associated with messages and builds the store record for persistence.

STORE CREATE

- Check required fields
- Check table constraints
- Gets the primary address type for a 'ST'ore Module
- Verify no details were passed in with message, none allowed for store create message.
- Check if a like store was passed in. If it is, then the price store and cost location must match the like store. If a like store was not passed in, the copy replenishment, activity, and delivery indicators must be No or null.
- Check that the store number passed in is not currently being used for a store or warehouse. Because the API adds to a staging table, validation must occur on the base tables that no record exists. Note that stores and warehouses in RMS cannot have the same unique identifier.
- Verify the start order days are greater than or equal to zero.
- Populate record with message data
- Populate the address detail table
- Default required record fields, not required on message.

STORE MODIFY

- Check required fields
- Check table constraints
- Verify the store exists on the base table
- Verify the start order days are greater than or equal to zero.
- Gets the address primary key for the store
- Populate record with message data.
- Populate the address detail table
- Check if the store's district was changed.
- Default required record fields, not required on message.

LOCATION TRAIT CREATE

- Verify store exists on the base table
- Verify location trait(s) were passed in on message
- Populate record with message data.

WALKTHROUGH STORE CREATE

- Verify store exists on the base table
- Verify walkthrough store(s) were passed in on message
- Populate record with message data.

LOCATION TRAIT DELETE

- Verify store exists on the base table
- Verify walkthrough store(s) were passed in on message
- Populate record with message data.

WALKTHROUGH STORE DELETE

- Verify store exists on the base table
- Verify walkthrough store(s) were passed in on message
- Populate record with message data.

STORE DELETE

- Verify store exists on the base table
- Verify a store can be deleted. Note that this check performs the same validation that occurs before a store is actually deleted in the batch process.
- Populate record with message data.

Bulk or Single DML Module

All insert, update and delete SQL statements are located in the family package. This package is STORE_SQL. The private functions in RMSSUB_STORE_SQL will call this package.

Filename: rmssub_xstoresqls/b.pls

```
RMSSUB_XSTORE_SQL.PERSIST_MESSAGE
               (O_error_message       IN OUT         VARCHAR2,
                I_store_rec           IN             STORE_SQL. STORE_ROW_TYPE,
                I_message_type        IN             VARCHAR2,)
```

This function determines what type of database transaction it will call based on the message type.

STORE CREATE

- Create messages get added to the staging table to be processed in a batch cycle. The address on the message is inserted as the primary address for the primary address type in the ADDR table. No other detail (child) processing occurs for creates.

STORE MODIFY

- Modify messages directly update the store table with changes. The address on the message is updated in the ADDR table. If the stores district has changed, the location traits from the old district will be removed, and the location traits for the new district will be added.

LOCATION TRAIT CREATE

- Adds location trait(s) to the store

WALKTHROUGH CREATE

- Adds walkthrough store(s) to the store.

LOCATION TRAIT DELETE

- Removes location trait(s) to the store

WALKTHROUGH DELETE

- Removes walkthrough store(s) to the store.

STORE DELETE

- Store gets added to a purging table to be processed in a batch cycle.

Message DTD

Here are the filenames that correspond with each message type. Please consult the RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Type	Message Type Description	DTD Name
XStoreCre	External Store Create	XStoreDesc.dtd
XStoreDel	External Store Delete	XStoreRef.dtd
XStoreLocTrtCre	External Store Location Trait Create	XStoreDesc.dtd
XStoreLocTrtDel	External Store Location Trait Delete	XStoreRef.dtd
XStoreMod	External Store Modification	XStoreDesc.dtd
XStoreWTCre	External Walk Through Store Create	XStoreDesc.dtd
XStoreWTDel	External Walk Through Store Delete	XStoreRef.dtd

Performance/Volume Considerations

Store creation and maintenance should have very low volume. Therefore, extensive tuning should not be required.

Design Assumptions

Required fields are shown in the Oracle Retail Integration Bus (RIB) documentation. The RMS feature for multiple addresses for stores is not supported in this API. The address in the message becomes the primary address for the primary address type of the store.

Tables

TABLE	SELECT	INSERT	UPDATE	DELETE
STORE_ADD	No	Yes	No	No
STORE	Yes	No	Yes	No
ADDR	Yes	Yes	Yes	No
DAILY_PURGE	No	Yes	No	No
LOC_TRAITS_MATRIX	Yes	Yes	No	Yes
SYSTEM_OPTIONS	Yes	No	No	No
TSF_ENTITY	Yes	No	No	No
WH	Yes	No	No	No
WALK_THROUGH_STORE	No	Yes	No	Yes

TABLE	SELECT	INSERT	UPDATE	DELETE
LOC_DISTRICT_TRAITS	Yes	No	No	No

Transfer

Functional Area

Transfer subscription

Design Overview

RMS subscribes to transfers from external subsystems that provide only basic information about these transactions, namely item, supplier, location and quantity.

It will be expected that RMS users will be responsible for updating and monitoring the financial and execution data such as transfer costs.

RMS monitors all of the shipments and receipts and will close the transfer once the received quantity equals the quantity requested or an outside system cancels the outstanding quantity. RMS will maintain the perpetual inventory for each location as it currently does.

The transfer RIB API will have defaulting logic which the API uses to populated defaulted fields. This is designed so that multiple sources can use the transfer API without having to conform to the same default values. Retailers can set-up their own set of default values or logic without having to modify the API code. For fields that are exposed on the message, if a value is provided, it will be used. Default values will only be used if a value is not provided on the message.

Consume Module

Filename: rmssub_xtsfs/b.pls

```
RMSSUB_XTSF.CONSUME
           (O_status_code     IN OUT              VARCHAR2,
            O_error_message   IN OUT              RTK_ERRORS.RTK_TEXT%TYPE,
            I_message         IN                  RIB_OBJECT,
            I_message_type    IN                  VARCHAR2)
```

This procedure will need to initially ensure that the passed in message type is a valid type for transfer messages.

If the message type is invalid, a status of “E” should be returned to the external system along with an appropriate error message informing the external system that the status is invalid.

If the message type is valid, the generic RIB_OBJECT needs to be downcast to the actual object using the Oracle’s treat function. There will be an object type that corresponds with each message type. If the downcast fails, a status of “E” is returned to the external system along with an appropriate error message informing the external system that the object passed in is invalid.

If the downcast is successful, then consume needs to verify that the message passes all of the RMS business validation. It calls the RMSSUB_XTSF_VALIDATE.CHECK_MESSAGE function to determine whether the message is valid. If the message passed RMS business validation, then the function returns true, otherwise it returns false. If the message fails RMS business validation, a status of “E” is returned to the external system along with the error message returned from the CHECK_MESSAGE function.

Once the message has passed RMS business validation, it is persisted to the RMS database. It calls the RMSSUB_XTSF_SQL.PERSIST() function. If the database persistence fails, the function returns false. A status of “E” is returned to the external system along with the error message returned from the PERSIST() function.

Once the message has been successfully persisted, there is nothing more for the consume procedure to do. A success status, “S”, is returned to the external system indicating that the message has been successfully received and persisted to the RMS database.

Business Validation Module

Filename: rmssub_xtsfvals/b.pls

It should be noted that some of the business validation is referential or involves uniqueness. This validation is handled automatically by the referential integrity constraints and the unique indexes implemented on the database.

RMSSUB_XTSF_VALIDATE.CHECK_MESSAGE

This overloaded function performs all business validation associated with create/modify messages and builds the transfer API record with default values for persistence in the transfer related tables. Any invalid records passed at any time results in message failure.

Like other APIs, the transfer API expects a snapshot of the record on both a header modify and a detail modify message, instead of only the fields that are changed. For a detail create or a detail modify message, only the TSF number will be validated at the header level; all other header fields are ignored.

TRANSFER CREATE

- Check required fields.
- Validate fields.
- Default fields (status at header,freight type and tsf type)
- Build transfer records.

TRANSFER MODIFY

- Check required fields on the header nodes.
- Verify TSF number already exists.
- Validate fields.
- Populate record.

TRANSFER DETAIL CREATE

- Check required fields on the detail node.
- Verify TSF number already exists.
- Verify tsf/item/loc does **not** already exist.
- Create item/loc relation if not already exists, including creating ITEM_LOC_SOH, ITEM_SUPP_COUNTRY_LOC, and PRICE_HIST records. If a pack item is involved, these records will be created for all component items.
- Populate record.

TRANSFER DETAIL MODIFY

- Check required fields on the detail node.
- Verify transfer/item/loc already exists.
- If TSF quantity is reduced, verify the new quantity is not below what has already been received plus what is being shipped or expected.
- Populate record.

RMSSUB_XTSF_VALIDATE.CHECK_MESSAGE

This overloaded function performs all business validation associated with delete messages and builds the transfer API record with default values for persistence in the transfer related tables. Any invalid records passed at any time results in message failure.

TRANSFER DELETE

- Check required fields.
- Verify TSF number already exists.
- Verify that TSF is not already shipped or received.
- Populate record for delete.

TRANSFER DETAIL DELETE

- Check required fields.
- Verify TSF/item/loc already exists.
- Verify that TSF line is not already shipped or received.
- Populate record with the TSF no/item/location for delete.

Bulk or single DML module

Filename: rmssub_xtsfs/b.pls

```
RMSSUB_XTSF_SQL.PERSIST
               (O_error_message      IN OUT         RTK_ERRORS.RTK_TEXT%TYPE,
                I_tsf_rec            IN             RMSSUB_XTSF.TSF_REC,
                I_message_type       IN             VARCHAR2)
```

This function checks the message type to route the object to the appropriate internal functions that perform DML insert, update and delete processes.

TRANSFER CREATE

- Inserts records in the TSFHEAD, TSFDETAIL, TSFDETAIL_CHRG tables.
- Updates records in the ITEM_LOC_SOH table.

TRANSFER MODIFY

- Updates a record in the TSFHEAD table.

TRANSFER DELETE

- Delete a transfer from TSFHEAD, TSFDETAIL, TSFDETAIL_CHRG tables.

TRANSFER DETAIL CREATE

- Inserts records in the TSFDETAIL, TSFDETAIL_CHRG tables.
- Updates records in the ITEM_LOC_SOH table.

TRANSFER DETAIL MODIFY

- Updates records in the TSFDETAIL, ITEM_LOC_SOH tables.

TRANSFER DETAIL DELETE

- Delete records from TSFDETAIL, TSFDETAIL_CHRG tables.

Message DTD

Here are the filenames that correspond with each message type. Please consult the RIB documentation for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
xtsfcre	Transfer Create Message	XTsfDesc.dtd
xtsfmod	Transfer Modify Message	XTsfDesc.dtd
xtsfdel	Transfer Delete Message	XTsfRef.dtd
xtsfdtlcre	Transfer Detail Create Message	XTsfDesc.dtd
xtsfdtlmod	Transfer Detail Modify Message	XTsfDesc.dtd
xtsfdtlcel	Transfer Detail Delete Message	XTsfRef.dtd

Design Assumptions

Required fields are shown in RIB documentation.

Tables

TABLE	SELECT	INSERT	UPDATE	DELETE
TSFHEAD	Yes	Yes	Yes	Yes
TSFDETAIL	Yes	Yes	Yes	Yes
TSFDETAIL_CHRG	Yes	Yes	Yes	Yes
ITEM_LOC	Yes	Yes	No	No
ITEM_LOC_SOH	Yes	Yes	No	No
PRICE_HIST	No	Yes	No	No
ITEM_MASTER	Yes	No	No	No
PACKITEM_BREAKOUT	Yes	No	No	No
STORE	Yes	No	No	No
WH	Yes	No	No	No
SYSTEM_OPTIONS	Yes	No	No	No

Vendor

Functional Area

Supplier

Business Overview

RMS subscribes to supplier information that is published from an external financial application. 'Vendor' can refer to either a partner or a supplier, but only supplier information is subscribed to by RMS. Supplier information also includes supplier addresses.

Processing includes a check for the appropriate financial application in RMS on the SYSTEM_OPTIONS table's FINANCIAL_AP column. The financial application (such as Oracle Financials) sends the information to RMS via the RIB.

Data Flow

An external system publishes a supplier type vendor, placing the supplier information onto the RIB (Oracle Retail Integration Bus). RMS subscribes to the supplier information as published from the RIB and places the information onto RMS tables depending upon the validity of the records enclosed within the message.

Message Structure:

The supplier message is a hierarchical message that consists of a supplier header record and a series of address records under the header record.

The header record contains information about the supplier as a whole. The address records identify the addresses associated with the supplier.

Package Impact

Subscribing to a supplier message entails the use of one public consume procedure. This procedure corresponds to the type of activity that can be done to a supplier record (in this case create/update).

Filename: rmssub_vendorcres/b.pls

Public API Procedures

```
RMSSUB_VENDORCRE.CONSUME
               (O_status_code        IN            OUT  VARCHAR2,
                O_error_message      IN            OUT  VARCHAR2,
                I_message            IN            CLOB);
```

This procedure accepts a XML file in the form of an Oracle CLOB data type from the RIB (I_message). This message contains a supplier message consisting of the aforementioned header and detail records. The procedure then places a call to the main RMSSUB_SUPPLIER.CONSUME function in order to validate the XML file format and, if successful, parses the values within the file through a series of calls to RIB_XML. The values extracted from the file are then passed on to private internal functions, which validate the values and place them on the supplier and address tables depending upon the success of the validation.

Private Internal Functions and Procedures (rmssub_vendorcre.pls):

Error Handling:

If an error occurs in this procedure, a call is placed to HANDLE_ERRORS in order to parse a complete error message and pass back a status to the RIB.

```
HANDLE_ERRORS
          (O_status                IN OUT              VARCHAR2,
           IO_error_message        IN OUT              VARCHAR2,
           I_cause                 IN                         VARCHAR2,
       I_program                   IN                         VARCHAR2);
```

This function is used to put error handling in one place in order to make future error handling enhancements easier to implement. All error handling in the internal RMSSUB_SUPPLIER package and all errors that occur during subscription in the RMSSUB_VENDORCRE package (and whatever packages it calls) flow through this function.

The function consists of a call to API_LIBRARY.HANDLE_ERRORS. API_LIBRARY.HANDLE_ERRORS accepts a program name, the cause of the error and potentially an unparsed error message if one has been created through a call to SQL_LIB.CREATE_MESSAGE. The function uses these input variables to parse a complete error message and pass back a status, depending upon the message and error type, back up through the consume function and up to the RIB.

Private Internal Functions and Procedures (other):

All of the following functions exist within RMSSUB_SUPPLIER.

Main Consume Function:

```
RMSSUB_SUPPLIER.CONSUME
              (O_status             OUT          VARCHAR2,
               O_error_message      OUT          VARCHAR2,
               I_document           IN           CLOB);
```

This procedure accepts a XML file in the form of an Oracle CLOB data type from the RIB (I_message) from the aforementioned public vendor procedure whenever a message is made available by the RIB. This message consists of the aforementioned header and detail records.

The procedure first gets the FINANCIAL_AP value from the system options table. It then validates the XML file format and, if successful, calls internal functions to parse the values within the file through a series of calls to RIB_XML. The values extracted from the file are then passed on to private internal functions, which validate the values and place them on the appropriate supplier and address database tables depending upon the success of the validation. The procedure then calls the CHECK_ADDR function to check that the proper addresses have been associated with the supplier.

PARSE_SUPPLIER

This function is used to extract the header level information from the supplier XML file and place that information onto an internal supplier header record.

The record is based upon the supplier table.

PARSE_ADDRESS

This function extracts the address level information from the supplier XML file and places that information onto an internal address record.

The record is based upon the address table:

PROCESS_SUPPLIER

After the values are parsed for a particular supplier record, RMSSUB_SUPPLIER.CONSUME calls this function, which in turn calls various functions inside RMSSUB_SUPPLIER in order to validate the values and place them on the appropriate supplier table depending upon the success of the validation. Either INSERT_SUPPLIER or UPDATE_SUPPLIER is called to actually insert or update the supplier table.

PROCESS_ADDRESS

After the values are parsed for a particular address record, RMSSUB_SUPPLIER.CONSUME calls this function. If the FINANCIAL_AP system option is set to '0', this function calls various functions inside RMSSUB_SUPPLIER in order to validate the values and place them on the appropriate address table depending upon the success of the validation. Either INSERT_ADDRESS or UPDATE_ADDRESS is called to actually insert or update the address table.

INSERT_SUPPLIER

This function first checks the UNIT_OPTIONS table to determine what the value of dept_level_orders is. If the dept_level_orders value is 'Y', the inv_mgmt_lvl is defaulted to 'D'. If the dept_level_orders value is anything other than 'Y', the inv_mgmt_lvl is set to 'S.'

The function then takes the information from the passed-in supplier record and inserts it into the SUPS table.

FUNCTION UPDATE_SUPPLIER

This function updates the SUPS table using the values contained in the I_supplier_record.

FUNCTION UPDATE_ADDRESS

This function updates the supplier information to the address table.

CHECK_CODES

The RMSSUB_SUPPLIER package, specifically the functions check_codes() and check_fkeys(), sends back descriptive error messages when codes are not valid or if a foreign key constraint is violated.

INSERT_ADDRESS

Insert supplier information to address table. If the address in the passed-in address record is the primary address for a particular supplier/address type, this function updates the current primary address so that it is no longer the primary.

VALIDATE_SUPPLIER_RECORD

Validate that all the necessary records are populated.

VALIDATE_ADDRESS_RECORD

Validate that all the necessary records are populated.

CHECK_NULLS

This function checks that the passed-in record variable is not null. If it is, it will return an error message.

Message DTD

Here are the filenames that correspond with each message type. Please consult Oracle Retail Integration Bus information for each message type in order to get a detailed picture of the composition of each message.

Message Types	Message Type Description	Document Type Definition (DTD)
VendorCre	Vendor Create Message	VendorDesc.dtd

Design Assumptions

- One of the primary assumptions in the current API approach is that ease of code will outweigh performance considerations. It is hoped that the 'trickle' nature of the flow of data will decrease the need to dwell on performance issues and instead allow developers to code in the easiest and most straight forward manner.
- The adaptor is only setup to call stored procedures, not stored functions. Any public program then needs to be a procedure.

Tables

TABLE	SELECT	INSERT	UPDATE	DELETE
SUPS	Yes	YES	Yes	NO
ADDR	Yes	YES	YES	NO
SYSTEM_OPTIONS	YES	NO	NO	NO
UNIT_OPTIONS	Yes	No	No	No
CODE_DETAIL	Yes	No	No	No

Work Order Status

Functional Area

Work Order Status Subscription

Design Overview

RMS subscribes to a work order status message sent from internal finishers. Work order status messages contain the items for which the activities have been completed along with the quantity that was completed. All items on transfers that pass through an internal finisher must have at least one work order activity performed upon them. When work order status messages are received for a particular item/quantity, it is assumed that all work order activities associated with the item/quantity have been completed. If work order activities involve item transformation or repacking, the work order status messages are always created in terms of the resultant item.

The work order status message is only necessary when the internal finisher and the final receiving location are in the same physical warehouse. If the internal finisher belongs to the receiving location, a book transfer is made between the internal finisher (which is held as a virtual warehouse) and the final receiving location (also a virtual warehouse). If the internal finisher belongs to the sending location's transfer entity, intercompany out and intercompany in transactions are recorded. Quantities on hand, reserved quantities, and weighted average costs are adjusted to accurately reflect the status of the stock.

Assume that 20 item 100 (White XL T-shirt) are sent to an internal finisher at the receiving physical warehouse where they will be dyed black, thereby transforming them into item 101 (Black XL T-shirt). If all finishing activities were successfully completed in this example, RMS could expect to receive a Work Order Status message containing item 101 with a quantity of 20.

It is possible to receive multiple Work Order Status messages for a particular item/transfer. Work order completion of partial quantities addresses the following scenarios:

1. Work order activities could not be performed for the entire quantity of a particular item at one time.
2. A given quantity of the particular item was damaged while work order activities were performed.

In terms of the previous example, RMS could receive a message containing item 101(Black XL T-shirt) with a quantity of 10. A message stating that work order activities were completed for the remaining 10 items could then be received at a later time.

The only scenario in which a Work Order Status message is necessary is when work order activities are taking place at an internal finisher that resides in the same physical warehouse as the transfer's final receiving location. This scenario can only take place in a multi-channel environment. In this scenario, the final 'leg' of the transfer will 'move' merchandise between two virtual warehouses in the same physical warehouse. As this movement cannot be done until all work order activities are completed for a specific item/quantity, the finisher must inform RMS of this completion.

Other finishing scenarios exist in which the finisher is not a virtual warehouse that shares a physical warehouse with the transfer's final receiving location. In these instances, Work Order Status messages are not necessary. This is because these scenarios dictate that merchandise must be physically shipped from the finisher to the transfer's final receiving location. RMS assumes that a finisher will not ship merchandise until all finishing activities have been completed for said merchandise. RMS will disregard Work Order Status messages sent in these scenarios.

Consume Module

Filename: rmssub_wostatuss/b.pls

```
PROCEDURE CONSUME
               (O_status_code         IN OUT         VARCHAR2,
                O_error_message       IN OUT         VARCHAR2,
                I_message             IN             RIB_OBJECT,
                I_message_type        IN             VARCHAR2)
```

This procedure is passed an Oracle Object, which it will validate to ensure all required data is present. It will ensure that the finisher and the transfer's final receiving location are in the same physical warehouse. If not, processing is deemed successful and halted. If the message contains an item, RMS work order complete processing will be called for that item. Otherwise, said processing will be called for all items on the transfer. If the entire transfer is processed, the child transfer (that is, the 'second leg') will be set to 'S'hipped status. Note that work orders are always associated with the second leg of multi-leg transfers. Whether processing is performed at the item or transfer level, transfer closing queue logic will be called to determine if the entire multi-leg transfer can be closed.

```
PROCEDURE HANDLE_ERRORS
               (O_status_code         IN OUT         VARCHAR2,
                IO_error_message      IN OUT         VARCHAR2,
                I_cause               IN             VARCHAR2,
                I_program             IN             VARCHAR2)
```

This is the standard error handling procedure that wraps the API_LIBRARY.HANDLE_ERROR function.

Message DTD

This subscription API consumes a message as specified in WOStatusDesc.dtd. See Oracle Retail Integration Bus documentation for details regarding the composition of a Work Order Status message.

Tables

TABLE	SELECT	INSERT	UPDATE	DELETE
TSFHEAD	Yes	No	Yes	No
TSF_DETAIL	Yes	No	Yes	No
TSF_ITEM_COST	Yes	No	Yes	No
DOC_CLOSE_QUEUE	No	Yes	No	No
ITEM_LOC_SOH	Yes	Yes	Yes	No
TRAN_DATA(VIEW)	No	Yes	No	No
INV_ADJ	No	Yes	No	No
INV_STATUS_QTY	No	Yes	Yes	Yes
INV_ADJ_REASON	Yes	No	No	No
V_PACKSKU_QTY	Yes	No	No	No
ITEM_LOC	Yes	No	No	No
ITEM_MASTER	Yes	No	No	No
INV_STATUS_CODES	Yes	No	No	No
SHIPSKU	Yes	No	No	No

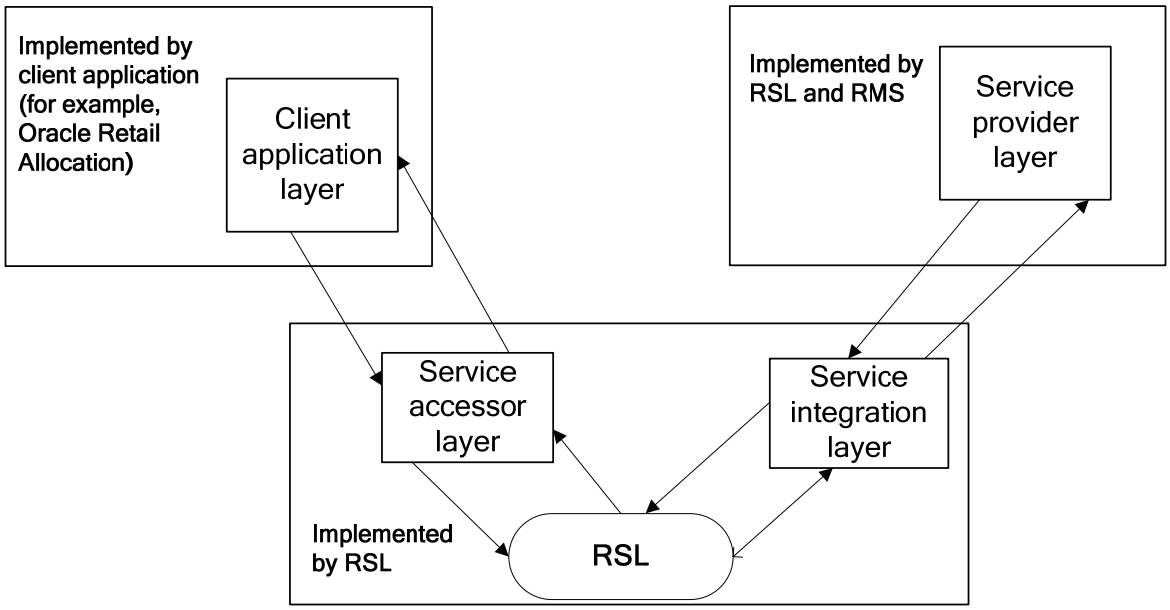
3
RSL for RMS

RMS and the Oracle Retail Service Layer (RSL)

RSL is a framework that allows Oracle Retail applications to expose APIs to other Oracle Retail applications. As shown in the diagram below, in RSL terms, there is a 'client application layer' and a 'service provider layer'. RMS includes the 'service provider layer' that owns the business logic.

The RMS implementation of RSL exposes a *synchronous* method to communicate with other applications (RIB-facilitated processing is asynchronous). All RSL services are contained within an interface offered by a Stateless Session Bean (SSB). To a client application, each service appears to be merely a method call.

For information about RSL-related configuration within the RMS application, see RSL documentation.



Client application and service provider processing through RSL

Functional Description of the Packages Used by RSL

The table below offers a functional description of the packages used by RSL.

Package	Description
RMSSVC_XLOCPOTSF	Through RSL, this call to RMS allows Oracle Retail Allocation to create/update a purchase order in RMS from a 'what if" allocation.